PROJECTE FINAL DE CARRERA

PRIVACY PRESERVING PHOTO SHARING ANALYSIS AND DERIVED COMPRESSION TECHNIQUE FOR IMAGES

Studies: Telecommunications Engineering

Author: Eric Bencomo Dixon

Director: Antonio Ortega

Year: 2013

Index

1.- Introduction 4
2.- Background 5
2.1.- JPEG Images 5
2.2.- P3 Algorithm and potential compression gain 9
2.3.- Linux Libjpeg library 11
3.- Motivation 12
3.1.- Visual and size effects of different thresholds on public image 12
3.2.- Effects of maintaining the sign of coefficients above threshold in the public part 13
3.3.- Finding the ideal threshold in terms of image quality and image size 14
3.4.- Practical implementation of the P3- Algorithm 15
3.5.- Differences in pixel value distribution for Original, Public and Secret images 16
3.6.- DCT coefficient distribution in public image 17
4.- Different proposed approaches 18
4.1.- All block dictionary 18
4.2.- Non-Zero Coefficient location dictionary (with and without sign) 20
4.3.- Run/size Block Dictionary: 20
5.- Implementation 23
5.1.- The Encoder 23
5.1.1.- Detailed encoding of the private part 25
5.2.- The Decoder 26
5.3.- Creation of the dictionary 27
5.4.- Results initial approach 27
6.- Implementation improvements 29
6.1.- Creating a dictionary for low frequency coefficients and Encoding the rest with JPEG. 29
6.2.- Using different thresholds for different coefficients in the image 30
6.3.- Decision of what blocks to introduce in the dictionary 32
6.4.- Optimization of jpeg encoding of high frequency coefficients (those not considered in the dictionary) 33
6.5.- Optimised Huffman table for low frequency AC DCT coefficient sizes 34
6.6.- Improved results 35
7.- Future Work 38
7.1.- Effects of varying the Scale Factor 38
7.2.- Defining ideal threshold within “length” level - Pros and cons of different alternatives 39
7.3.- Analysis of the private part 41
7.4.- Delta encoding – Finding the closest block in the dictionary 41
7.5.- Common image dictionary 42
8.- Conclusions 44
9.- Acknowledgements 45
10.- Bibliography 46
11.- Annex 47
11.1.- Images 47
11.2.- Additional figures 48

1.- Introduction

At the end of my Telecommunication Engineering studies I decided I wanted my thesis to focus on image processing. I found the UPC-USC exchange program and after being accepted started working with Professor Antonio Ortega.

When deciding the project theme I found an interest in the thesis done by a recently graduated PhD student (Moo-Ryong Ra), which involves an algorithm for privacy preserving in image sharing (P3- Privacy Preserving Photo Sharing) (explained in 2.2).

The initial idea for my thesis was to analyse the algorithm to: find possible ways to attack the privacy component, figure out ways to apply filtering to the privacy enabled images or reduce the overall size of the resulting files. After some weeks of research we found that the approach upon which the algorithm is based gives base to creating a new encoding system to improve the overall compression of jpeg images.

The scheme we implemented comes from the idea that the thresholding technique used in the P3 algorithm will make run-length blocks common enough within an image or even across multiple images to create dictionaries with information on a full block rather than a symbol for each non-zero coefficient (as jpeg does).

To summarise, in this project we study the P3 algorithm and a new encoding scheme for images based on its thresholding technique.

2.- Background

2.1.- JPEG Images

As of today, JPEG (Joint Photographic Experts Group) accounts for most of the lossy digital images that are stored in computers, cameras, phones and other devices and shared across the internet. The reason for this is that even if JPEG is a relatively old compression system with worse compression gains than other more modern systems, it is very fast, it has been around long enough to find any possible bugs and inconveniences and last but not least, it is cheap and simple enough to implement in digital cameras and other simple devices.

JPEG is a lossy encoding system, which means that in order to compress an image it loses some image data (ideally little enough that the human eye cannot tell the difference). We can say that JPEG has 6 main stages:

1) RGB to YCbCr colour space conversion.

2) Subsampling of the Chorminance (Cb & Cr) components (typical but not required).

3) Transformation into the frequency domain using the Discrete Cosine Transformation (DCT) in 8x8 coefficient blocks for each component.

4) Quantization of the 8x8 DCT blocks using a standard Quantization table (different for Y than for Cb & Cr) and a user defined Quality factor.

5) Coding of the DCT transformed blocks using a run/length technique – *Our implementation will be different from jpeg from this point onwards*

6) Huffman encoding of the run/size code.

In order to reduce the amount of information stored in the image, jpeg uses the fact that the human eye interprets spatial and acuity content better than colour, so it converts the RGB representation of an image to Luminance (Y) and Chrominance (Cb & Cr) components. The Luminance component has most of the spatial information of the image (it is a weighted average of the original Red, Green and Blue components for each pixel), while Chrominance-blue (Cb) and Chrominance-red (Cr) have very little spatial information. The transformation from RGB to YCbCr is the following:

$$\begin{bmatrix} Y & Cb & Cr \end{bmatrix} = \begin{bmatrix} R & G & B \end{bmatrix} \begin{bmatrix} 0.299 & -0.168935 & 0.499813 \\ 0.587 & -0.331665 & -0.418531 \\ 0.114 & 0.50059 & -0.081282 \end{bmatrix}$$

Once the image has been transformed to this colour space, and taking advantage of the fact that humans do not perceive changes in chrominance very effectively, jpeg can subsample the Cb and Cr components. Jpeg usually removes every other horizontal and vertical lines of pixels, resulting in subtracting ¾ of the information for each one of these components, or in other words, half the information in the image.

After this step, JPEG separates the image into 8x8 pixel blocks and converts them all to the frequency domain using a two-dimensional Discrete Cosine Transform (DCT):

$$G_{u,v} = \sum_{x=0}^{7}\sum_{y=0}^{7} \alpha(u)\alpha(v) g_{x,y} \cos\left[\frac{\pi}{8}\left(x+\frac{1}{2}\right)u\right] \cos\left[\frac{\pi}{8}\left(y+\frac{1}{2}\right)v\right]$$

$$\alpha(u) = \begin{cases} \sqrt{\frac{1}{8}}, & \text{if } u = 0 \\ \sqrt{\frac{2}{8}}, & \text{otherwise} \end{cases}$$

where:

- u is the horizontal spatial frequency, for the integers $0 \leq u < 8$
- v is the vertical spatial frequency, for the integers $0 \leq v < 8$
- $\alpha(u)$ is a normalizing scale factor to make the transformation orthonormal
- $g_{x,y}$ is the pixel value at coordinates (x, y)
- $G_{u,v}$ is the DCT coefficient at coordinates (u, v)

In order to reduce the size of the DC coefficient, jpeg subtracts 128 to all the pixel values in the image before transforming to the frequency domain (this is equivalent to subtracting 1024 to the DC value).

Once we have the 8x8 DCT coefficients we perform the "lossy" part of jpeg, which is the only step in which we lose image quality (other than the subsampling of the chroma components), but it is also the step that allows us to get the most compression. Considering that the distribution for different coefficients is not the same, the jpeg committee created different quantization tables for the Luminance and Chrominance components:

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Fig. 1.– Standard Luminance Quantization table.

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

Fig. 2.– Standard Chrominance Quantization table.

These quantization tables give us a fixed relation across different coefficients (as we can see, high frequency coefficients are divided by a much larger value than low frequency coefficients). The 8x8 DCT blocks are divided coefficient by coefficient by the value in the corresponding location of the quantization table and a scale factor (this scale factor defines how much of the information we discard, resulting for a higher factor in higher compression but less image quality).

$$DCT_{quantized} = round\left(\frac{DCT_{coefficients}}{Q \cdot Scale_{factor}}\right)$$

The scale factor is calculated with the next formula:

$$Scale_{factor} = \begin{cases} \dfrac{50}{Quality_{factor}} & if \quad Quality_{factor} < 50 \\ 2 - \dfrac{Quality_{factor}}{50} & if \quad Quality_{factor} \geq 50 \end{cases}$$

$$0 < Quality_{factor} < 100$$

The Quality factor is defined by the user and, as we can see from the formula above, ranges from 0 to 100 giving 100 the maximum quality and 0 the minimum. A very common Quality factor is 75.

Once the image has been Quantized and the resulting coefficients are rounded to the closest integer (trying to encode decimal numbers would be a lot more costly in bits) jpeg encodes the resulting 8x8 DCT coefficient blocks in the following manner:

- It calculates the difference between the current and previous DC coefficient and sends a Huffman encoded variable length symbol with the number of bits required to send

the coefficient value (using one's complement for negative numbers) followed by the actual value.

- Starting from the first AC coefficient and following a zig-zag order (Figure x), it counts the number of zeros prior to the next non-zero value and sends a Huffman encoded symbol that contains information on the number of zeros and the size in bits of the non-zero value (in the same way as it does for the DC coefficient), followed by the amplitude. This technique is called run length encoding.
- After the last non-zero value in the block jpeg sends an "End Of Block" (EOB) symbol, which lets us know that from that coefficient onwards there are only zeros. This is very useful because, after the quantization step, it is very likely to find zeros for high frequency coefficients, which means that by sending the EOB symbol we can avoid sending a lot of extra information.

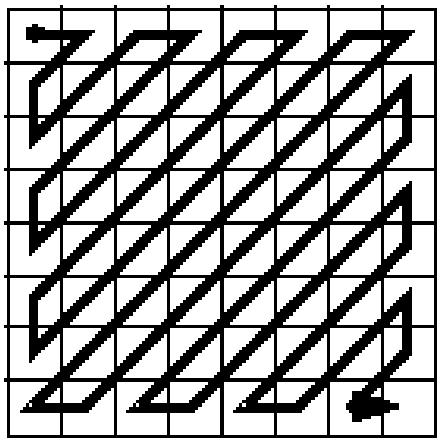

Fig. 3.– Zig-zag matrix.

2.2.- P3 Algorithm and potential compression gain

When studying the P3 algorithm we observed an interesting phenomenon; the algorithm separates an image into 2 files: one containing most of the visual information of the image and very little volume information (few bytes), and the other one containing very little visual information and most of the volume in bytes. Assuming that this separation meant that the part with most of the volume information contained similar jpeg encoded 8x8 DCT blocks inspired us to look for ways to improve the compression scheme for these components.

The Privacy Preserving Photo Sharing algorithm (PPIS) is a new system developed by PhD Moo-Ryong Ra and Professors Antonio Ortega and Ramesh Govindan that ensures photo privacy without sacrificing the latency, storage and bandwidth benefits provided by Photo sharing Service Providers (PSPs). The motivation behind the creation of this algorithm is the increasing need in privacy when sharing images through PSPs while keeping all the benefits these provide.

PSPs perform server-side transformations on images (such as resizing or cropping) in order to improve the latency of image downloads (a mobile device, for example, will receive a much smaller image than a device with a large high resolution screen). When sharing a photo, if we used traditional encryption techniques, the PSP would not be able to perform these server-side transformations. P3, however allows us to maintain all those benefits and at the same time not allowing algorithms like face detection and SIFT feature extraction to obtain information from the public image and allowing edge detection algorithms to only estimate a small fraction of pixels defining edges correctly.

Starting from a jpeg image as an input, the P3 algorithm separates it into a public and a secret image (both in jpeg format to allow a fast reconstruction of the original). As the name implies, the public image will be kept in the open and while containing most of the information in bytes, while containing very little visual information of the original file. Its counterpart the secret image will contain the extra information which will ideally account for a very small percentage of the size in bytes of the original file.

The P3 algorithm works as follows:

If we look at the JPEG encoding process (2.1), after the quantization step we have an image separated in blocks of 8x8 DCT coefficients. The DC coefficient for each block is the value for frequency 0, which means it contains the information for the average of the 8x8 block. In order to distort the image visualization as much as possible while maintaining as much of the information as we can in the open (public image), the P3 algorithm removes this coefficient and stores it in the private part. As this is not enough for most images, for the AC coefficients the algorithm defines a threshold and, for any coefficients above that threshold (or below –threshold) it stores the difference between the coefficient and the threshold in the private part. Any values that are sent to the private part are stored in the same position of the same DCT 8x8 block maintaining jpeg format for a speedy reconstruction.

After doing this we will have two separate images. One of them, containing most of the byte information in the original image but with very little visual information from it (we should only

be able to see a grey picture with some edge information). The other image will be small relative to the original file and will contain the key information the public image is missing to recover the original file. Both images can now be sent (one of them in the open and the other encrypted or through a secure channel) to whoever we want to share it with without having to worry about undesired individuals laying their eyes on them.

It is worth mentioning that a higher threshold means more information in the public part (good) but on the downside it also allows us to visualize more of the original image (bad), so we have to find a balance between level of privacy and percentage of the image in the public part. To understand why it is important to take this factor into account we should remember that any information that goes to the secret part cannot be processed, so the bigger portion of the original image that goes to the secret part the fewer benefits we get from processing the public part for an improved user experience.

As we mentioned, the advantage of this scheme compared to encrypting the whole image is that it allows us to send most of the information in the open and as a jpeg image, allowing PSPs to process them. One example of this is that when we upload a picture to Facebook; they send different resolutions of the same image to different devices. A phone will get a much smaller file than a laptop, which means higher transmission speeds and less data costs. Encrypting the whole image might be safer than the P3 algorithm, but it loses a lot of the advantages of using PSPs.

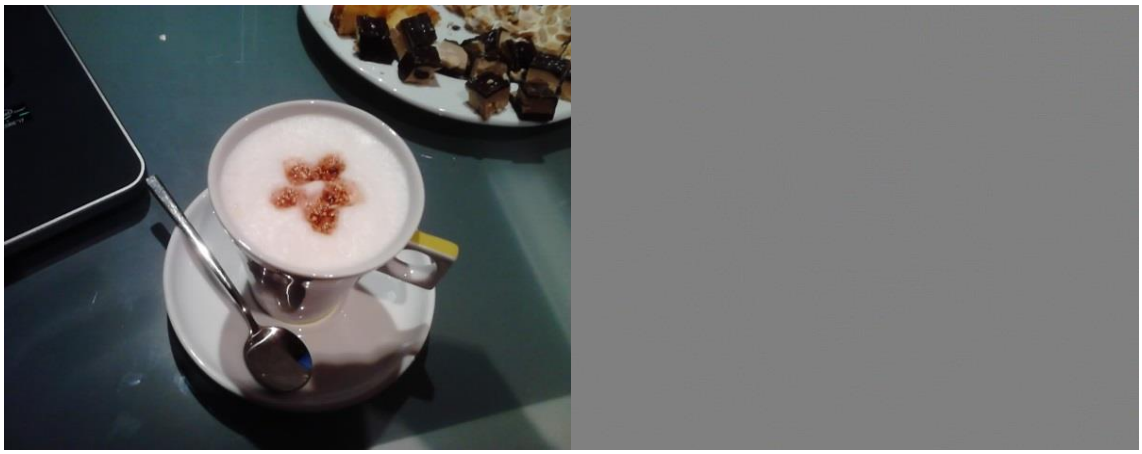

Fig. 4.– Secret & Public images - Threshold 5 - Secret: 102Kb – Public: 246Kb.

As we can see, even if the secret image contains most of the visual information, it is much smaller than the public image. For higher thresholds the secret image becomes even smaller, the disadvantage being as we mentioned that the public image contains more visual information.

The combined size of the public and private images is always larger than the original image. However, the fact that we are changing any value above threshold to threshold leads us to believe that the 8x8 DCT coefficient blocks in the public image will become more common, so it is reasonable to assume that a more efficient encoding scheme than jpeg can be found to store them.

2.3.- Linux Libjpeg library

In order to implement our proposed encoding scheme we chose to modify an existing jpeg library to take advantage of its existing functionalities.

The Linux Libjpeg library is an open source project written in C that implements JPEG encoding/decoding alongside several other utilities to manipulate jpeg images. The library is maintained by the Independent JPEG group (IJG) and it is one of the main open source jpeg libraries. It has been used by many companies and was key to the success of the jpeg standard. The library was first publicly released in October 1991, and has undergone a lot of development since then (the latest stable version being v9).

3.- Motivation

At the beginning of the project we run several studies on p3 encoded images to get an intuition of the effects of the algorithm on image data and the effects of varying the threshold or other components on the overall image privacy:

3.1.- Visual and size effects of different thresholds on public image

First, we analyse the visual effects that varying the threshold has on the public image:

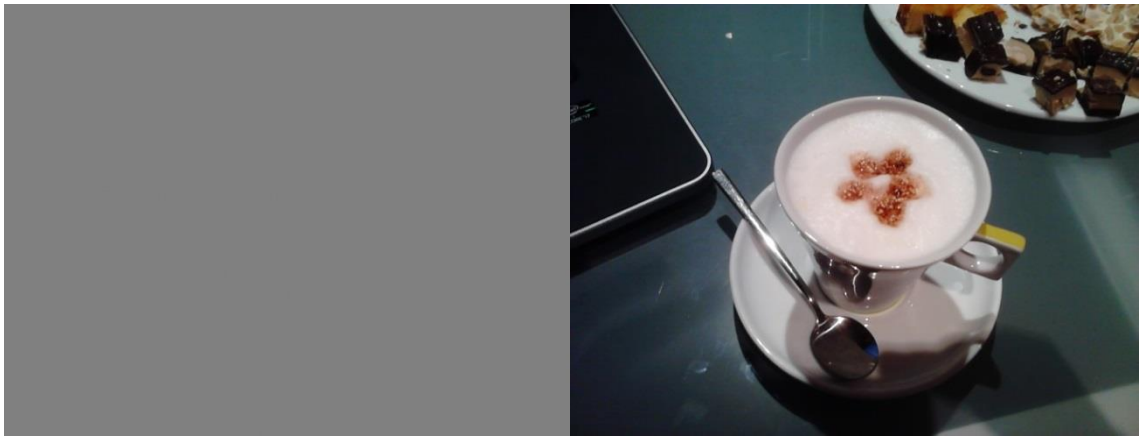

Fig. 5 – Public (left) & Secret (right) Images-Threshold: 1-Q=75-Public: 165KB - Secret: 184KB.

Fig. 6 – Public (left) and Secret (right) Images – Threshold: 20 – Public: 243KB - Secret: 73KB.

As we can see from the public image, changing the threshold from 1 to 20 has a big effect both on the size of the public-secret images and on visual privacy. For threshold 1 the public image contains almost no visual information, but on the downside it is 32% smaller than the public image for threshold 20, which means that a lot more information went to the secret part (the secret part for threshold 1 is 60% larger than for threshold 20). We can see then that the decision of what threshold to use is not unimportant.

3.2.- Effects of maintaining the sign of coefficients above threshold in the public part

If instead of changing any coefficient value above threshold or below –threshold to threshold we modified the value in absolute terms but did not change the sign we would observe the following:

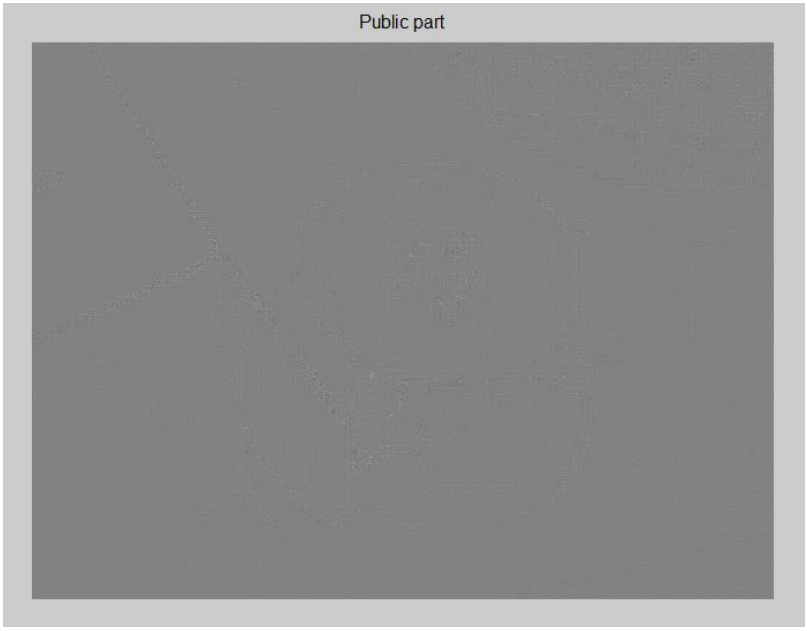

Fig. 7 – Threshold = 1 - Sign removed from public part.

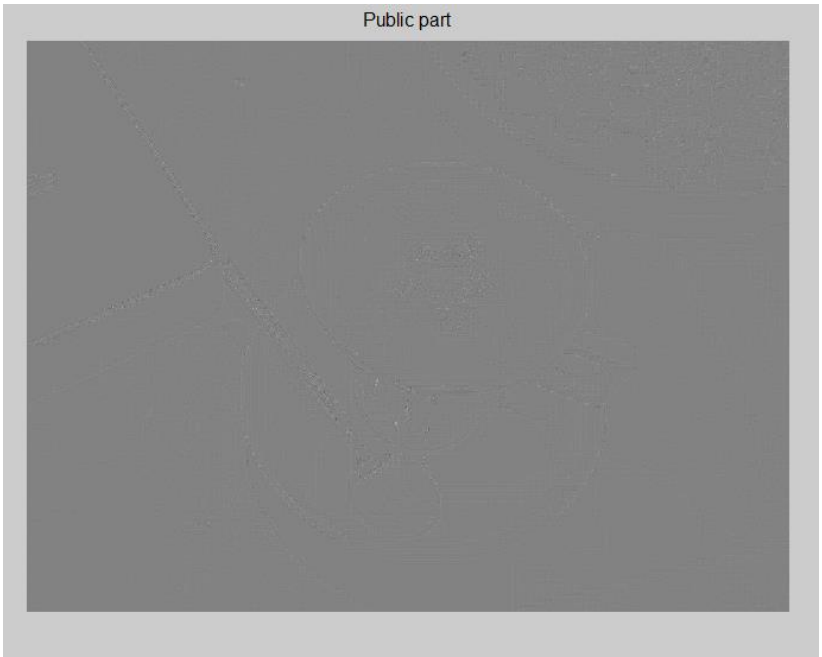

Fig. 8 – Threshold = 1, Sign maintained in public part. We can see the edges of the image.

At a first glance it might be hard to see, but if we look with close attention it is clear that if we maintain the sign in the public image it will be much easier to visualize the original image from the public part. We can conclude then that keeping the sign of coefficients above threshold in the private part is an important factor to ensure image privacy.

3.3.- Finding the ideal threshold in terms of image quality and image size

As we explained, the P3 algorithm focuses on privacy, which means that our main goal is to ensure that the public image is as unrecognizable as possible while keeping most of the information in the open. However, once that is achieved we also want to make the combination of public/private images to be as small in size as we can, which means that choosing the ideal threshold requires some analysis (We should remember that in order to achieve good privacy while maintaining the advantages offered by PSPs our goal is to have a threshold of as small a value as possible while maximizing the public image size).

In order to understand how the threshold value affects the size of the public image, we study the effect different thresholds have on image sizes:

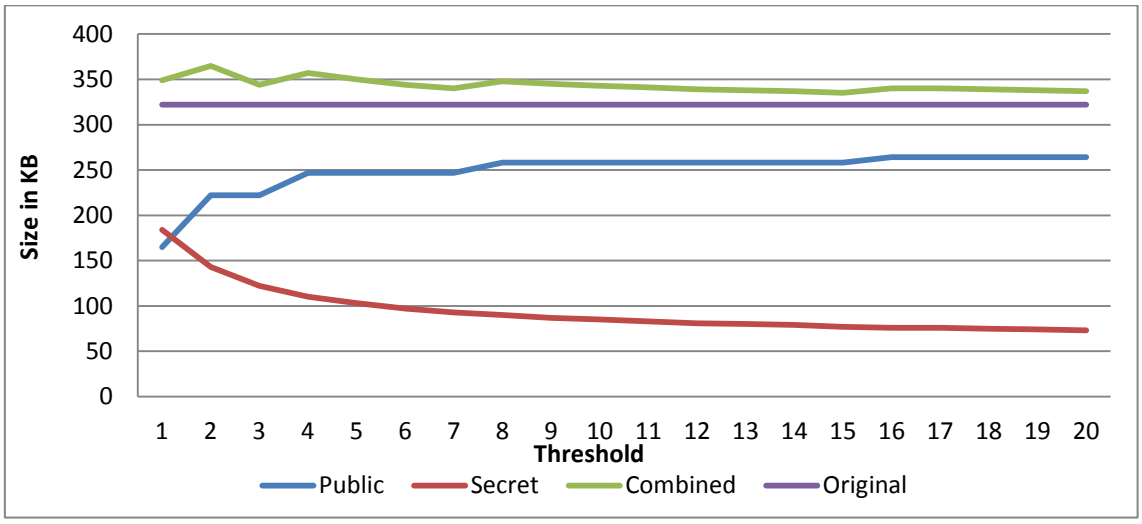

Fig. 9 – Bytes required for image Coffee cup for different thresholds – Q=75.

As we can see, the public image size is the same for some threshold values and jumps up at specific locations (the jumps in size get smaller for bigger thresholds because of the Laplacian distribution of AC DCT coefficients. This distribution results in higher coefficients being less likely to appear, which means that every time we increase the threshold we will be adding less information to the public part). In order to understand this we must go back to jpeg encoding; when we encode a jpeg image, the AC coefficients are encoded as run/length with a Huffman symbol, followed by the number of bits specified in the "length" part (these bits tell us the coefficient value). There are several coefficient values with the same number of bits, resulting in all the threshold values included in the same "length" to require the same amount of bits to be written. That means that for threshold values included in the same length we should have the same public image size.

If we study the values included in each "length" we will see how the changes in size coincides with the changes in the number of bits required to write the threshold value, which occur every threshold = 2^k with k = 0, 1, 2, 3, 4…

length (size of the coefficient in bits)	values
1	-1,1
2	-3,-2,2,3
3	-7…-4,4…7
4	-15…-8,8…15
5	-31…-16,16…31
6	-63…-32,32…63
7	-127…-64,64…127
8	-255…-128,128…255
9	-511…-256,256…511
10	-1023…-512,512…1023

Fig. 10 – Values included in each DCT coefficient size.

As we assumed, the public image grows every time the length of the coefficient increases. On the other hand, the higher the threshold value the smaller the secret image will be (due to the fact that we are sending more information to the public part).

We can conclude then, that in order to minimize the overall size of the public and private images, we should always choose a threshold value as close to the largest number for that "length" as possible. By choosing the highest value, we will make the secret image smaller (or in the worst case equal in size) while maintaining the size of the public image.

3.4.- Practical implementation of the P3- Algorithm

Parallel to this study we worked on a practical implementation of the P3 algorithm on Android and an iOS applications. In order to share the privacy enabled image we modified the libjpeg implementation of the p3 algorithm by Moo-Ryong Ra to add the functionality of encrypting the secret image with a Symmetric key and introducing it in the APP7 marker of the public image as explained in [1].

The idea for this implementation is that the user who wants to share or store an image can combine the public and secret images to send them as one single file to a server. The server has access to the public image and can apply any transformations necessary without having access to the secret part. The user can then keep the image stored in the server safe from undesired access or decide to share it with other individuals. In that case, the user can share the symmetric key with his or her friends by encrypting it with the other individual's Public RSA key.

We are currently working to launch an iOS application with an implementation of the P3 algorithm.

3.5.- Differences in pixel value distribution for Original, public and secret images

In order to get an idea of what the effect of separating the images in the visual domain was, we decided to study the histogram for the three images:

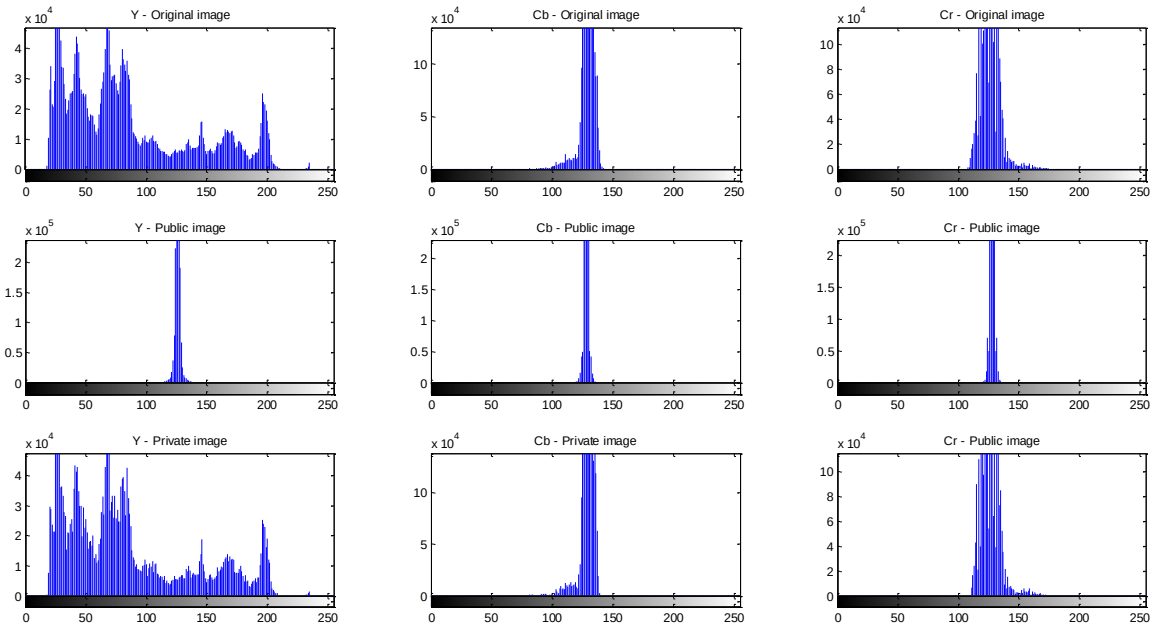

Fig. 11 – Original, Public & Private image Histograms.

As we can see from the figure, the Cb and Cr components are somewhat similar for the three cases (being the possible values much more concentrated for the public image than the other 2). However, for the luminance component there is a distinguishable difference in the distribution of pixel values. Where the original and secret images seem to have an unclear distribution, the public image has all its coefficients centered around the middle value (~128) with a very narrow Laplacian or Gaussian distribution. The reason for this distribution can be attributed to removing the DC coefficient.

These results seem to indicate that blocks in the public image are a lot more similar than in the original image (at least in the spatial domain). In order to get a better feeling on the effect of applying the p3 algorithm on public images we decided to study the coefficient distribution in the frequency domain for the public part.

3.6.- DCT coefficient distribution in public image

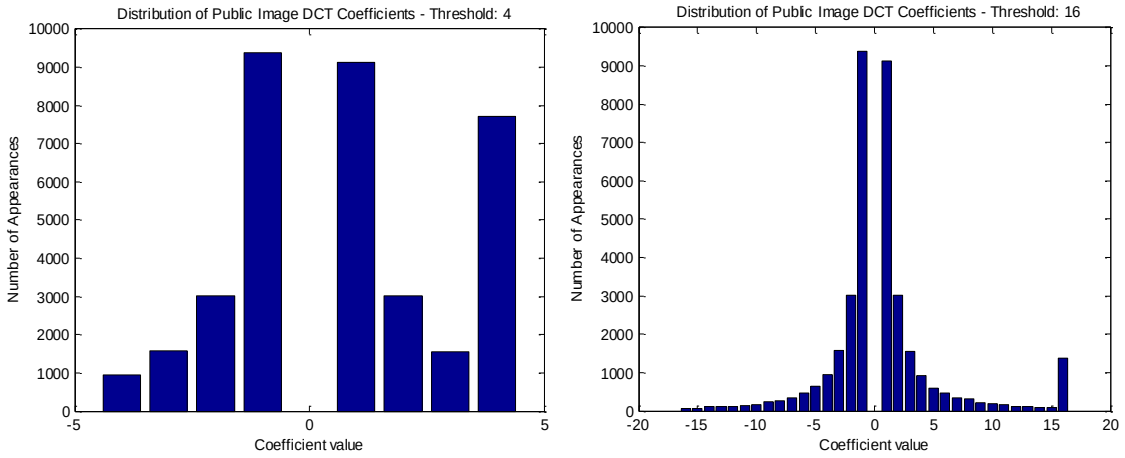

Fig. 12 –DCT coefficient distribution public image - Luminance – Threshold: 4 (left) & 16 (right)

As we can see, as a result of applying the p3 algorithm, the public image has a different coefficient distribution than the original file. What used to be a Laplacian distribution has now been truncated changing all the values in the tail (for both negative and positive values) to threshold. The fact that all the combinations of unlikely coefficient values (high values are less likely to occur than low values) are set to threshold in the public part means that the rare combinations of blocks we would find in the original image now become part of more common combinations. Having less block combinations in an image, each with higher probability of appearance would mean that a more efficient encoding system than jpeg could be used to get better compression on the public image. This would compensate the increase in size of the combined public and private images compared to the original file.

With that in mind we decided to investigate different new encoding techniques for the public part.

4.- Different proposed approaches

Taking advantage of the fact that the blocks in the public image have less combinations than those in a normal image, we set ourselves to create a dictionary that would contain in each symbol information relevant to the whole block rather than only the number of zeros preceding a non-zero value followed by the value and an end of block to notify that the remaining coefficients are all zero.

With these intentions we came up with three different ideas with which to define the symbols in the dictionary. In order to estimate the number of bits we would need to send each symbol we calculated the entropy of each resulting block (different for each approach) and rounded the number up to the closest larger integer (the closest integer to the right) (we decided to round up like that to be somewhat pessimistic):

$$Entropy = H(X) = -\sum_{i=1}^{n} p(x_i) \cdot \log_2 p(x_i)$$

$$Symbol\ size = ceil(Entropy)$$

We run our initial tests on a 600x600 image with Quality factor = 75 (11.1, fig. 31). Different image sizes and quality levels might show very different results (we will leave that analysis for future work).

4.1.- All block dictionary

As the name implies, this approach involves creating a symbol for every single 8x8 DCT coefficient combination of values in the image. If we created a dictionary for every single image and didn't consider the overhead caused by having to send the dictionary alongside the image, this approach would be the ideal solution, as it would send the most repeated block with the least number of bits possible.

In this approach every symbol would have all the information in the 8x8 DCT block, so in order to create an estimate of the file sizes we compared the number of bits we would need to send for the image data with jpeg encoding to the symbol size found for each block based on its probability:

$$Bits_{jpeg} = \sum run/size_{symbol} + EOB + \sum amplitude_{non-zero\ coefs}$$

$$Bits_{allblock} = Symbol\ size$$

$$Gain = \frac{Bits_{jpeg} - Bits_{allblock}}{Bits_{jpeg}}$$

Using this scheme we obtained the following results:

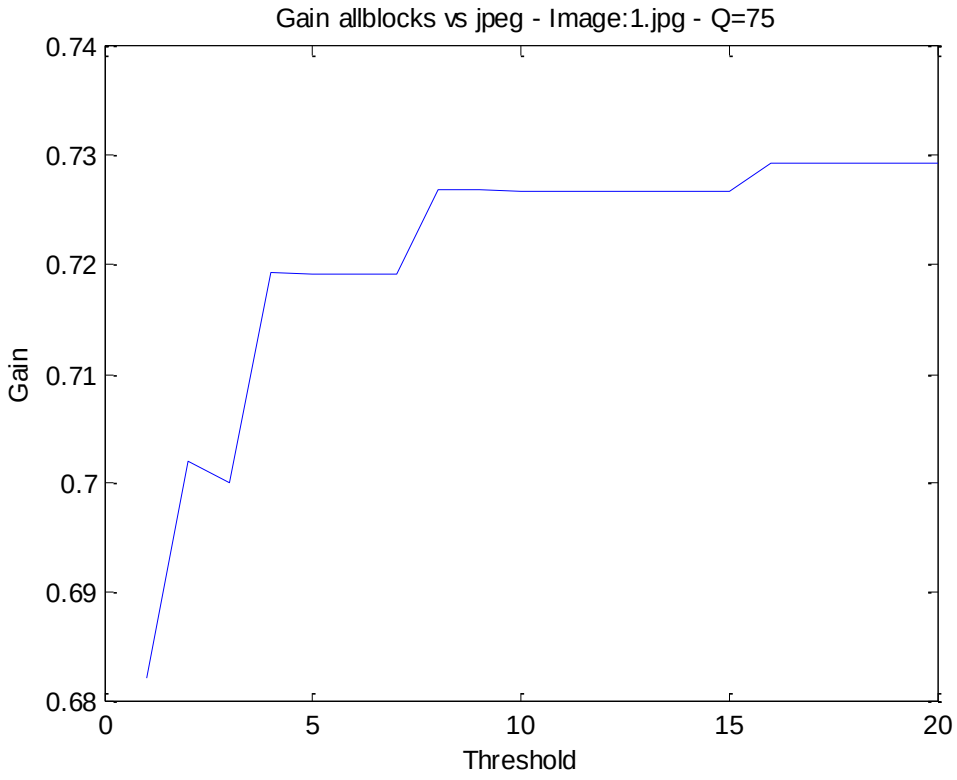

Fig. 13 – Gain allblocks vs jpeg – img:1.jpg – Q=75

These results show that in an ideal world, if we sent the minimum number of bits (based on the block's entropy) for each 8x8 DCT coefficient block we would get gains between 68% and 73% for this specific public image. This was to be expected as with this approach we are sending all the information in each entire block with the minimum number of bits possible. We could use these results then as an upper bound of the gain we can achieve by separating the information in 8x8 DCT blocks (there is no way we could send the same information with less bits than with this scheme).

The problem is that, on one hand, even with the Laplacian distribution of the DCT coefficients and the thresholding we performed, if we consider all the possible combinations of 8x8 DCT coefficient blocks in an image:

$$Combinations = (possible\ coefficient\ values)^{64}$$

With so many combinations it will be very unlikely to find blocks that are repeated within an image, so the creation of a dictionary becomes useless.

On the other hand, we could propose to create a dictionary that had all the possible combinations of blocks, but again, if it had to contain all the possible combinations of 8x8 DCT coefficients it would have such a large number of entries that we would require many more bits than jpeg to encode each block.

We can conclude then, that this approach would give us the best compression gain but has, at least with our scheme, a non-realistic implementation.

4.2.- Non-Zero Coefficient location dictionary (with and without sign)

This approach involves creating a symbol for each combination of non-zero DCT AC coefficient values in an 8x8 block. The practical advantage of this approach is that, as it only stores the location of non-zero coefficients it increases the chances of seeing repeated blocks and requires a much smaller dictionary than the previous case. The disadvantage is that alongside the symbol for the location of the non-zero coefficient values we have to send the actual value, which is trickier than one would assume; we would have to send a variable length symbol for the number of bits required for the value followed by the value itself (in a similar way as jpeg does for DC coefficients). If we don't want to have a fixed length symbol when sending the value of a non-zero coefficient (it would be very inefficient to have to send 10 bits for each coefficient if there is only one coefficient in the image that requires 10 bits), we need a variable length symbol to tell us the length in bits of the following non-zero coefficient. Doing that is clearly more inefficient than sending a single symbol for all the information in the image, but we have the advantage that this approach might realistically be used to create a dictionary that contains a large enough number of blocks for the same symbol.

For this approach we considered two scenarios, keeping the sign in the symbol and sending it with the value. Sending the sign information with the symbol would incur in more combinations of blocks (bad) and less information to send with the value (good). Sending the sign information with the value would cause the opposite. If we decided to implement this scheme we would have to weigh in the advantages and disadvantages of each option.

4.3.- Run/size Block Dictionary:

Our third approach (and the one we implemented) is a hybrid between the last 2 cases. As we saw that the allblocks approach does not give us a common enough set of blocks, in the same way as jpeg does, we figured that we could use the fact that multiple values are contained within the same "length", or in other words, require the same number of bits to be represented. With that in mind we created a dictionary containing in each symbol the location of each non-zero coefficient in the block and the number of bits required for that coefficient.

Similar to what we did for the allblocks scheme, we estimated the gain we would get for the public image (without considering the dictionary) in the following way (The symbol size is calculated with the entropy of the run/size encoded blocks):

$$Bits_{jpeg} = \sum run/size_{symbol} + EOB + \sum amplitude_{non-zero\ coefs}$$

$$Bits_{Hybrid} = Symbol\ size + \sum amplitude_{non-zero\ coefs}$$

$$Gain = \frac{Bits_{jpeg} - Bits_{allblock}}{Bits_{jpeg}}$$

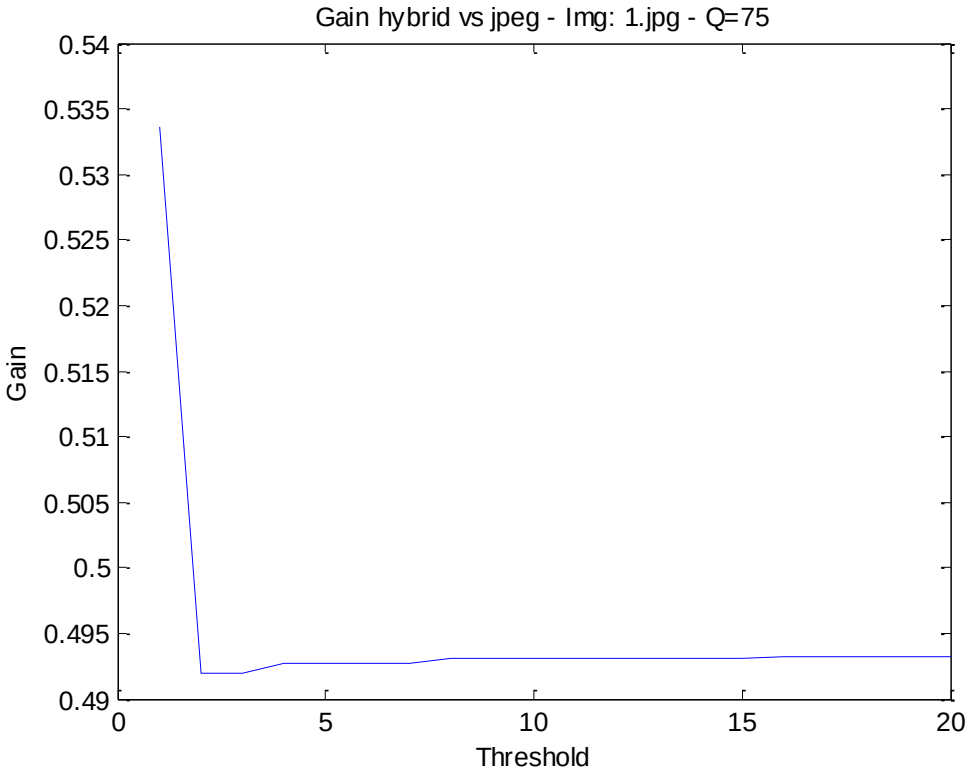

Fig. 14 – Gain Hybrid vs jpeg – img: 1.jpg – Q=75

As we expected, the gain we get for this case is lower than what we obtained for the allblocks encoding (as we explained before, the gain we got for that case can never be beaten when sending 8x8 DCT blocks). However, we still get very good gains on the public image (around 50%), and we have the advantage that our blocks will be more common than in the allblocks case, which can be translated to a more realistic implementation.

We should consider that depending on how we choose to create the dictionary there is going to be a number of blocks that are not worth including. If we create a dictionary for each image, for example, we will find that some blocks only occur once or too little times to compensate for the overhead of adding them to the dictionary.

To get a better intuition on what happens block by block we plotted the number of bits we would require to send each public block with jpeg and compared it to the number of bits we would require with our proposed scheme:

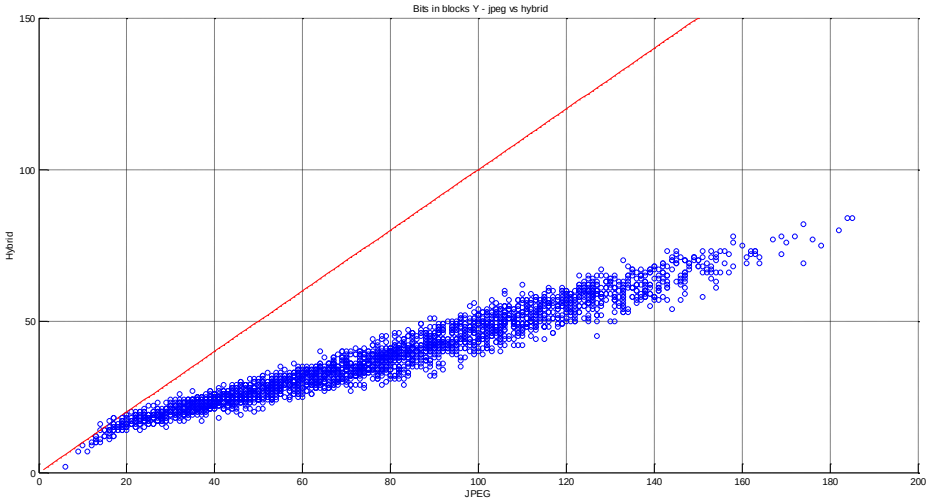

Fig. 15 – Bits/block public image jpeg vs hybrid - Luminance - img:1.jpg – Thld = 16

As we can see, for the luminance component results are very promising. There is a very small number of blocks for which we would need more bits with our proposed scheme than with jpeg. We should note, however, that the blocks for which jpeg performs worse are the least likely to occur.

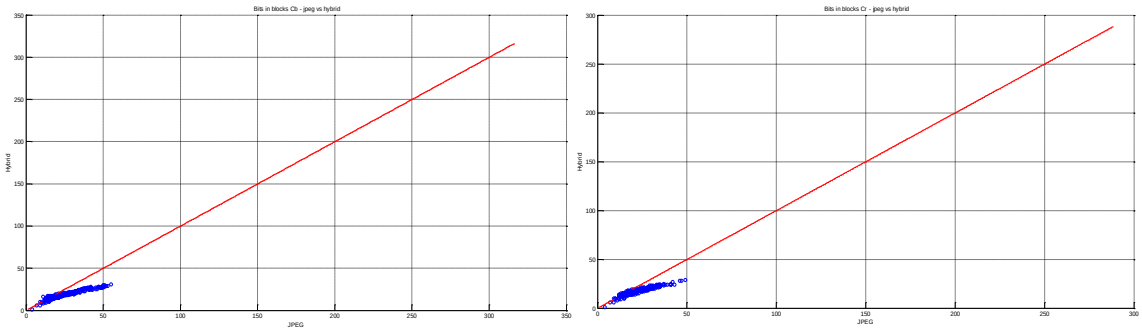

Fig. 16 – Bits/block public image jpeg vs hybrid – Cb (left) & Cr (right) - img: 1.jpg – Thld = 16

In the case of Cb and Cr, as these components have a higher level of quantization, they are more prone to have fewer and lower non-zero coefficients than the Luminance component. Even if that means that there are less block combinations in the image (resulting in less bits per symbol in our dictionary), as jpeg performs better with lower and fewer coefficients, there will be a smaller difference between the number of bits required to send the block information with our scheme compared to jpeg (We can clearly see how both our symbols and jpeg's symbols require on average a much lower number of bits for the Cb and Cr components than for Y).

The results we obtained for this scheme were indeed very promising, so we decided to try and implement a system that not only considered the public image, but also the secret part. It will obviously be more difficult to beat jpeg's compression scheme in this case, but we believe that these results give ground to trying.

5.- Implementation

As we mention in 4.3, the results we obtained when comparing our compression scheme to jpeg for public images were very promising, so we decided to try and get better compression gains across the whole image rather than only the public part. From now on our main focus will be compression gain, so the decision of which threshold to use will be driven by compression results and not privacy concerns. For ease of understanding we will keep the nomenclature of public and private parts.

At this point we decided to implement a real system based on the hybrid encoding scheme we proposed in 4.3 (with some modifications to consider the DC coefficient and the information above threshold we were previously ignoring) and explore the results considering all aspects of the scheme. We developed our encoder/decoder by modifying the existing Independent JPEG group's libjpeg C library (2.3).

Conceptually, the difference between this scheme and jpeg is that instead of sending a symbol containing information for the size of each non-zero coefficient in a block and the number of zeros preceding it, we send only one symbol that contains the same run/length information for the whole block (except for coefficients above threshold, for which we only keep the size of the threshold; we will have to send the difference between coefficient and threshold in the private part). This scheme will work well if our images have block combinations that occur very often.

5.1.- The Encoder

The scheme of the encoder is as follows:

We used existing libjpeg code to:

- *Open an image*
- *Extract the 8x8 DCT coefficient blocks.*
- *Do jpeg encoding of blocks not found in the dictionary*
- *Write output streams to file.*

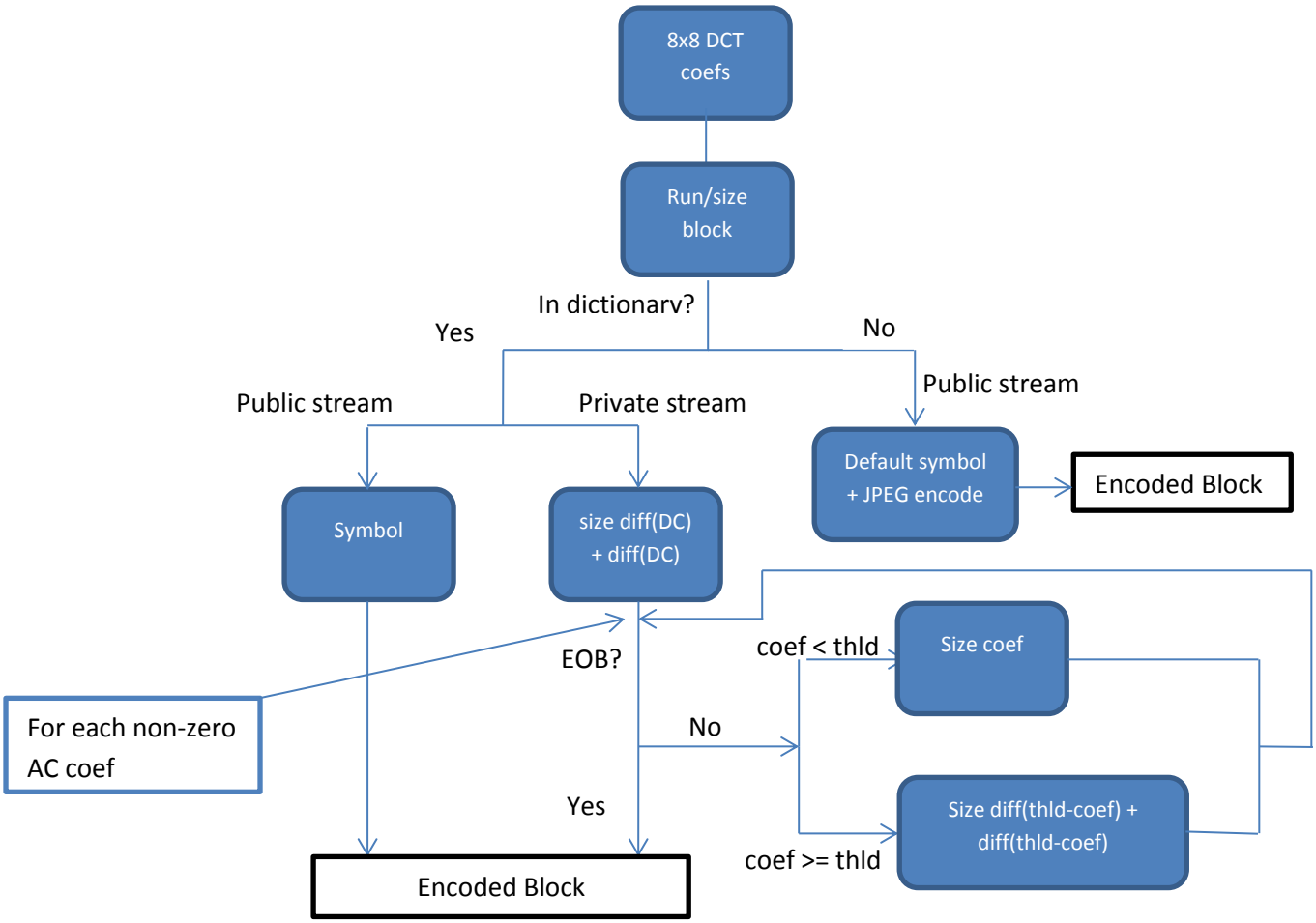

Fig. 18 – Encoding Scheme.

Our new encoding scheme does the following:

- Create 2 output streams. We will call them public and private.
- Obtain the run/size encoding of non-zero AC coefficients in the block reducing the size of values above a given threshold (that means above threshold or below –threshold) to the size of the threshold (we will call this public block)
- Look for the public block in an existing dictionary (either created specifically for the image, for a set of images or for all images)
- At this point there are 2 possibilities:

    - The block is found in the dictionary. In this case, we write:
        - Public part:
            - The symbol for the given block found in the dictionary
        - Private part:
            - Size of DC coefficient (Huffman symbol)
            - Amplitude of DC coefficient (one's complement for negative numbers)
            - Amplitude of AC coefficients below threshold (one's complement for negative numbers)

        - Size of AC coefficients with absolute value equal or above threshold (Huffman symbol)
        - Amplitude of AC coefficients with absolute value equal or above threshold (one's complement for negative numbers)
    - The block is not found in the dictionary. In this case we write:
        - Public part:
            - Symbol for "default block" (block not found in dictionary)
            - Jpeg encoding of the original block
        - Private part:
            - No data

5.1.1.- Detailed encoding of the private part

As we mentioned in 5.1 there are a few things to take in account for the private part of the encoded image in this scheme. On one hand, we will only have data in the private part if a block is not found in the dictionary. On the other hand, if a block is found in the dictionary, the DC coefficient will always be encoded in the same way:

- Size of the difference between the current and previous DC coefficients encoded with JPEG´s Huffman table for DC coefficients (different tables for Y than Cb & Cr). As we are sending the coefficient in the same way as JPEG does, the tools jpeg uses have already been optimized for the task.
- Amplitude of the difference between the current and previous DC coefficients using one's complement for negative values

There are however several scenarios for the AC coefficients:

-thld < coef < thld: In this case, the public symbol contains information on the position of the coefficient and its size in bits, so we only have to send the actual value/amplitude (we use the same one's complement scheme jpeg uses)

coef < -thld or **coef > thld:** In this case, the public symbol contains the location of the coefficient and the threshold value, but not the coefficient's size, so we will have to send 2 symbols: one for the size of the difference between the coefficient and the threshold (Huffman encoded) and one for the amplitude of the difference between the threshold and the DCT value.

Coef == -thld or **coef == thld:** This case is a bit of an exception. In essence it is the same as the previous case, but as the difference between threshold and coefficient is 0 in both cases we have to specify the sign of the coefficient. To do that we came up with a simple solution: If the coef == thld we will send size 0 followed by another 0 (to signify positive thld). In the other case we will send size 0 followed by a 1 (to signify negative thld). In other words, we will send the symbol for size 0 followed by an extra bit to tell the decoder whether it is a positive or a negative value.

An important factor to take into account is that for this scheme to work the threshold can only be a multiple of 2^k with k=1,2,3… or, said differently, the smallest element for a specific length (fig. 10). We selected this scheme because we considered that by choosing the smallest number in a specific size as threshold, whenever we see that specific size we will automatically know that the value in question is equal or above threshold. In this way, for coefficients equal or above threshold we do not have to send the threshold value followed by the difference between coefficient and value; we only send the difference. In 7.2 we will discuss the advantages and disadvantages of this decision.

5.2.- The Decoder

In order to implement the decoder we also took advantage of the existing libjpeg C linux library.

The decoder scheme is as follows:

- Open public output stream:
- Read symbol –> 2 possibilities:
    - Default symbol:
        - We call libjpeg's jpeg decoder to obtain the 8x8 DCT coefficients (the block is written in the public file right after the default block symbol)
    - Other symbols:
        - Obtain Run/size block from dictionary
        - Open Private output stream
            - Get DC coefficient
            - For the non-zero AC coefficients:
                - Size(coef) < size(thld): Read the coefficient value
                - Otherwise:
                    - Read Huffman encoded size of difference between coefficient & threshold (or – threshold if coefficient is negative)
                    - Read value of the difference.
                    - Add difference to threshold or subtract from – threshold to recover the original coefficient
- At this stage we have recovered all the original 8x8 DCT coefficients in the image and we can call libjpeg existing functionalities to encode the image in jpeg format.

5.3.- Creation of the dictionary

Implementation wise, the creation of the dictionary uses functionalities we created in the libjpeg library to obtain all the thresholded run/size blocks in an image, and python code to count the number of appearances of each block, organize them in order of appearance and decide whether the block should be included in the dictionary or not. The probability of all the blocks that are not to be considered is added up and used for the default symbol (this symbol will tell our encoder/decoder that what follows is a jpeg encoded block). At this point, where we know all the blocks that are to be considered for the dictionary, their probability of appearance and the probability of the default symbol, we call a third python program to create a Huffman dictionary for our run/length encoded blocks. In our dictionary we save the run/length encoded blocks using jpeg encoding (we believe it is the best solution as it is already optimized for that purpose and we already have the jpeg DC and AC Huffman tables).

Our implementation can create a dictionary for a single image or by considering the blocks of as many images as we decide and can make multiple considerations when deciding what blocks to include in the dictionary (more on this in 6.3). For the initial test on our implementation, however, we studied the results for a per-image dictionary considering only the number of times a block appears in the image to decide whether to include it in the dictionary or not.

5.4.- Results initial approach

For a per-image dictionary, we plotted the gain we obtained from encoding image 1_baseline.jpg. The y axis shows the size in Bytes and the x axis shows the number of times a block had to appear in the image to be considered in the dictionary. From left to right and top to bottom we analysed thresholds 1,2,4,8 and 16:

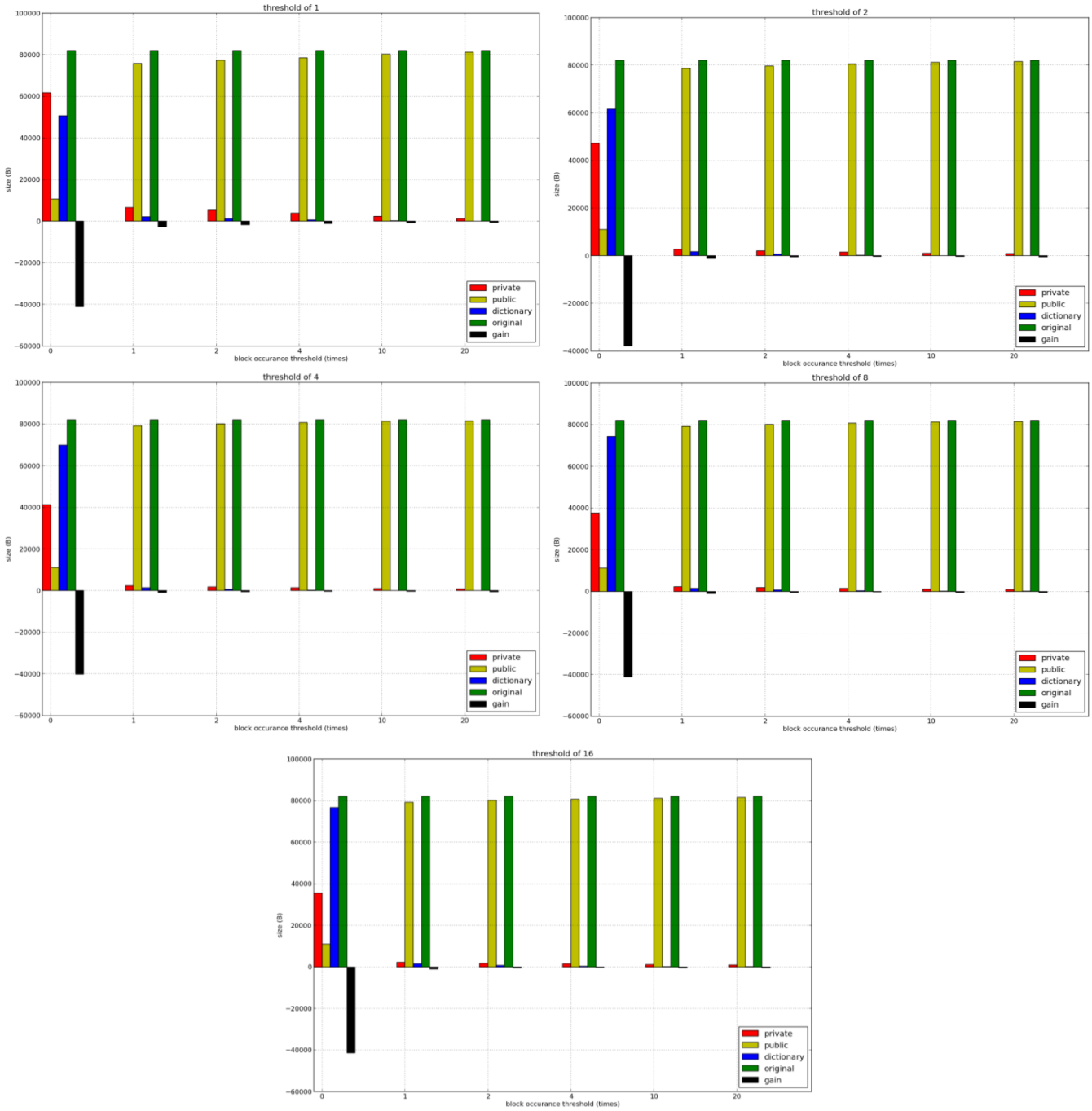

Fig. 19 – Gain – per-image dictionary – 1_baseline.jpg – Different thlds – Q = 75 (Y axis: Size in Bytes, X axis: number of times a block has to occur to be in the dictionary)-

From these results we can get several ideas. First, the fact that the dictionary is very large if we consider blocks that occur only once and very small otherwise indicates that with this scheme the image has a very large number of blocks that occur only once, which is bad when trying to create a per-image dictionary unless the blocks that occurred more than once accounted for most of the information in the image. However, as the size of the public image is in this case very close to the original and the size of the private image is very small we can assume that we have a proportionally small number of blocks that made it to the dictionary. In order to understand this better we decided to study the distribution of block repetitions (see 5.1).

With these results in mind we decided to look for ways to improve our implementation and make our blocks more common.

6.- Implementation improvements

As we found that our initial implementation was not giving us blocks that were repeated enough across a single image to create a per-image dictionary we set ourselves to try and make those blocks more common with the following approaches:

6.1.- Creating a dictionary for low frequency coefficients and Encoding the rest with JPEG

As jpeg images have nonzero values most likely around the low frequency coefficients we decided to create our dictionary considering only the first N coefficients (starting from the lowest frequency and going up following the zig-zag scheme). We first analysed, for one image, the weight of the blocks being repeated specific numbers of times (for example, how many of the blocks happened only once, how many happened twice, etc.) considering in each case a different number of DCT coefficients for the dictionary.

The results we got are as follows (Y axis: Number of blocks that appear “x” times, X axis: Number of appearances of block in image):

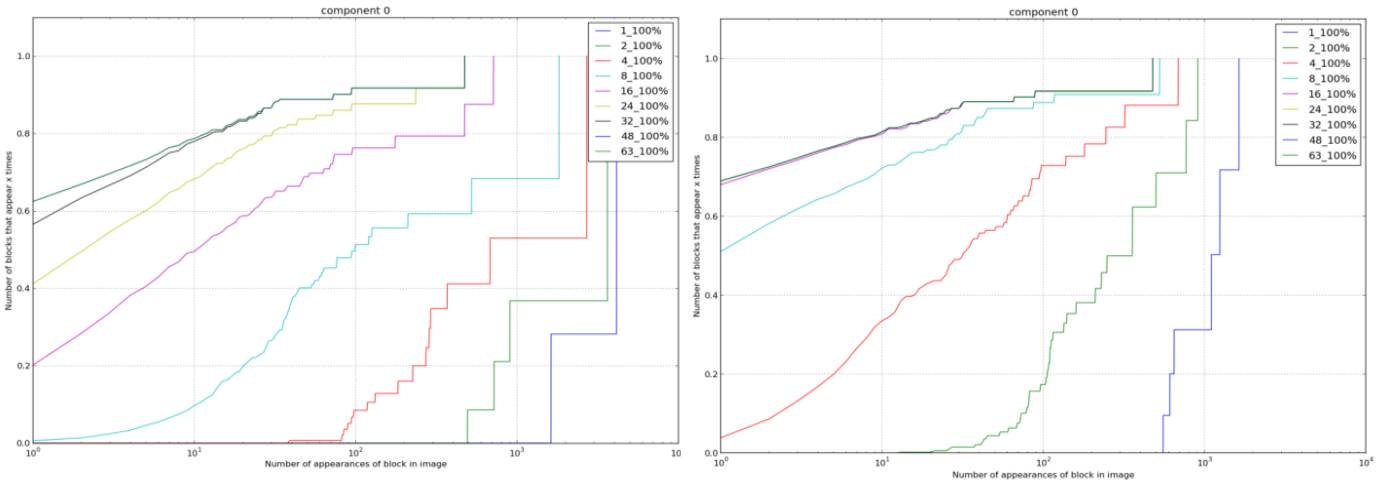

Fig. 20 – Block repetitions for different thld masks– img: 1_baseline.jpg – Thld = 1 (left) & 16 (right) - Q=75 – Component: Luminance (Y).

As we were expecting, for the luminance component (component 0) even with a threshold of 1, if we consider all the coefficients to create the dictionary more than 60% of the blocks occur only once, and only around 10% occur more than 100 times, which is telling us that our original approach for a single image is not good enough to get consistent gains. If we look at the cdf plot in more detail we can see how with a mask of 16 bits around 50% of the blocks are repeated more than 10 times which seems to be more reasonable. For threshold 16 we would even have to consider fewer coefficients to get similar results (around 6). If we considered all the coefficients about 70% of the blocks would occur only once.

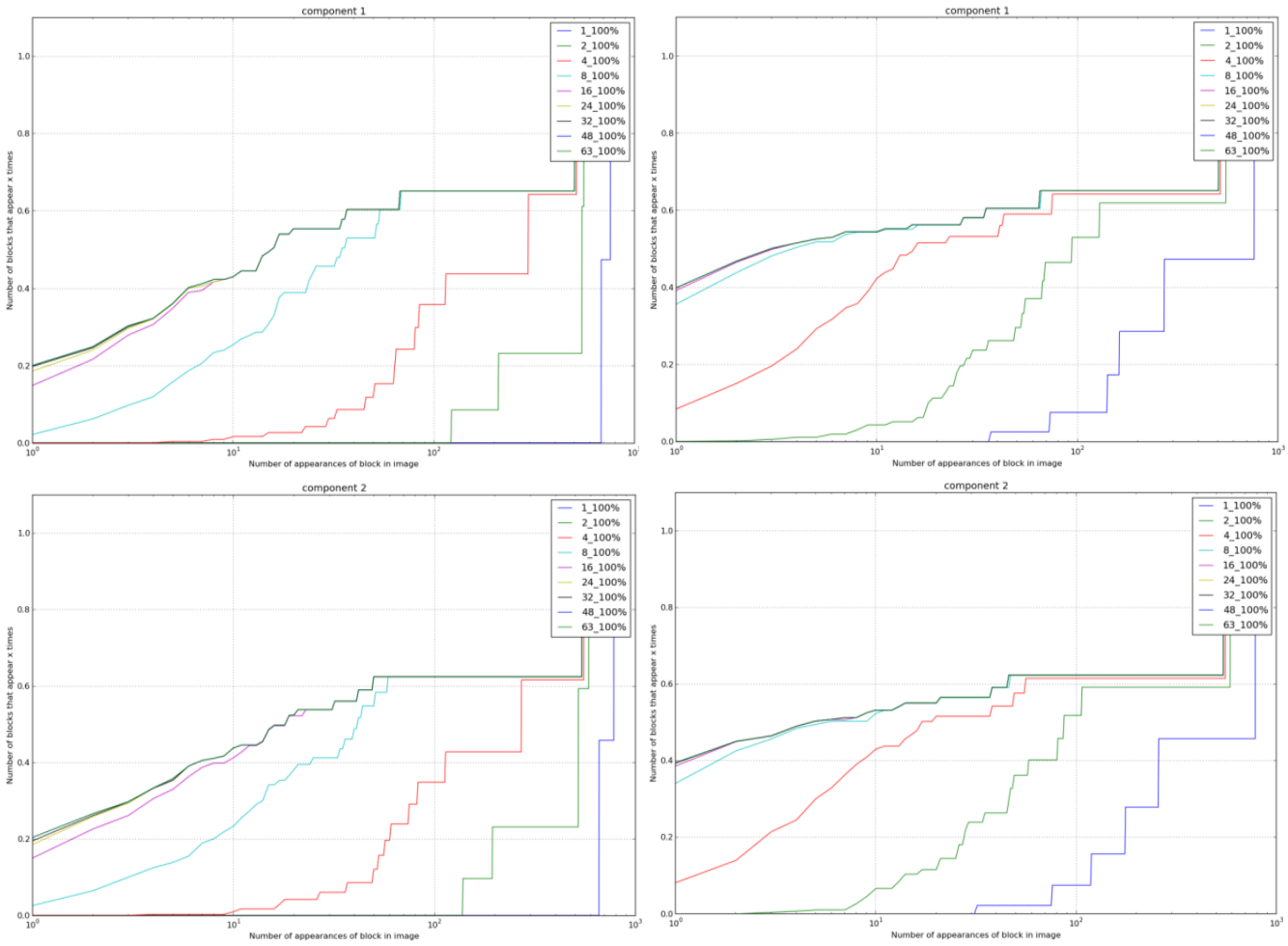

Fig. 21 – Block repetitions for different thld masks in Image 1_baseline.jpg – Thld = 1 (left) & 16 (right) - Q=75 – Components: Cb (top) & Cr (bottom).

Opposed to the analogous case, for components 1 & 2 (Cb & Cr) we can say that with low thresholds it is reasonable to consider the whole block in the dictionary. It is important to take into account that if we don't consider the whole block when creating the dictionary we will need to send the high frequency coefficients in some other way, incurring in more bits to be sent (even if all the high coefficients are always zero we will have to send an EOB with each block), so whenever possible we should use all the coefficients to create the dictionary. For threshold 16 we still get that around 50% of the blocks occur more than 10 times, which is an indicator that for these components our original approach might be better than considering only a number of coefficients for the dictionary.

6.2.- Using different thresholds for different coefficients in the image

In the same way that we analysed only considering the low frequency coefficients for the dictionary, we decided to study the effect of having different thresholds for different coefficients. As it is much more likely to get higher non-zero coefficients for low frequency locations than for high-frequency we will use higher thresholds for these locations (we will use thresholds in decreasing size from lowest to highest frequency). With a mask of 8 coefficients and thresholds ranging from 16 to 0 in decrements of 2 we obtained the following:

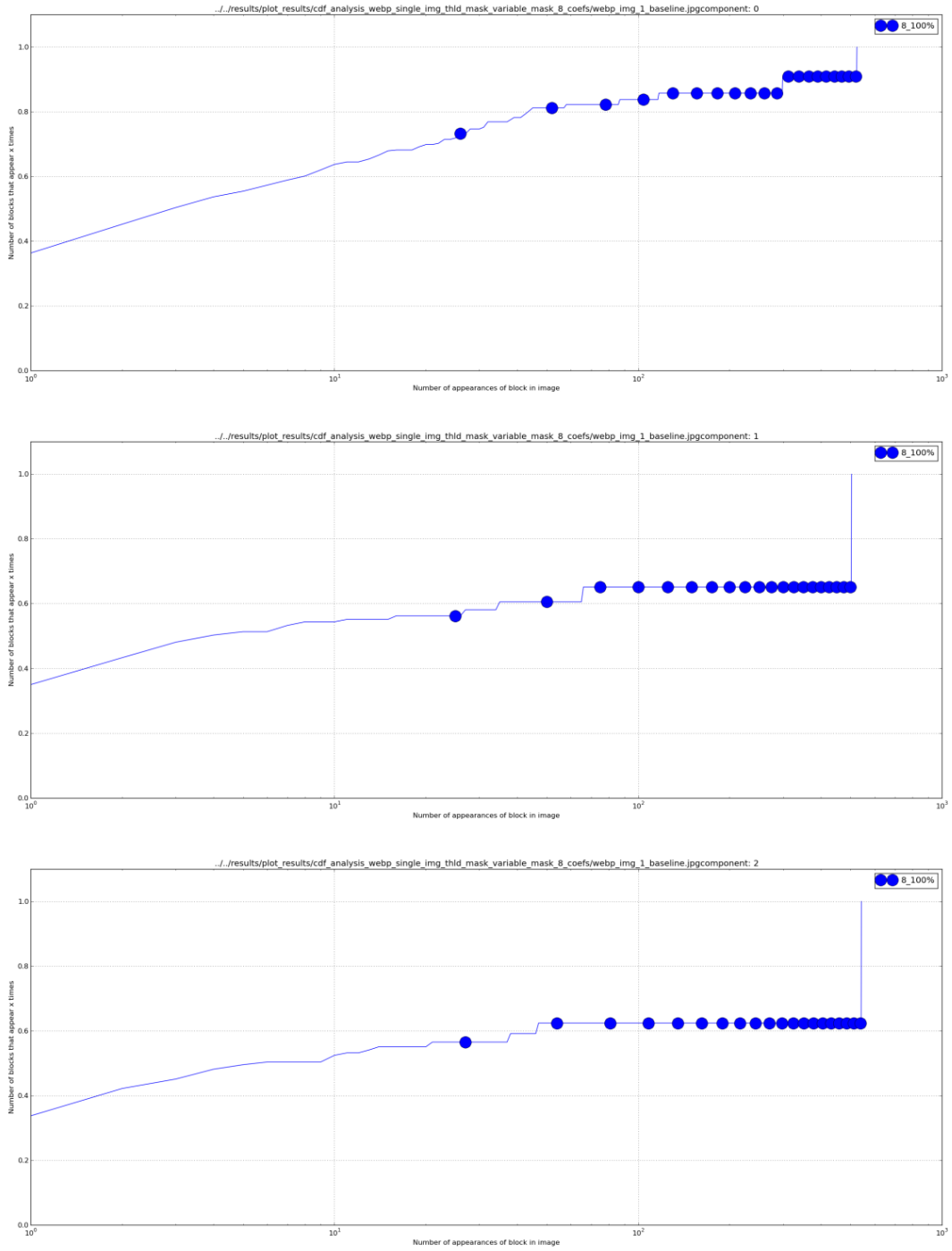

Fig. 22 – Block repetitions different thlds – thld mask: 8 – Components: Y (top),Cb (middle),Cr (bottom) – img: 1_baseline.jpg.

As we can see from the results, by using different thresholds we obtain that for Luminance under 40% of the blocks are repeated only once and about 35% occur more than 10 times. This could result in a considerable gain compared to jpeg if the symbol lengths of our dictionary symbols are small compared to the run/size jpeg symbols (which might be tough if a block has very few non-zero coefficients).

6.3.- Decision of what blocks to introduce in the dictionary

There are two reasons that make our scheme better than jpeg:

1) Number of bits required for our symbol smaller than the number required for jpeg.

2) Block occurring enough times in an image to compensate for the cost of adding the entry to the dictionary.

With that in mind, in order to decide whether a block goes in the dictionary or is discarded we follow two rules. First, as we already did for our initial tests, we only introduce entries in the dictionary for blocks that occur at least a specific number of times (if a block appears only once, for example, it would be more efficient to encode it using jpeg than to create an entry in the dictionary and then using the symbol. In the first case we would only have to send the jpeg encoding once versus sending the jpeg encoding once for the entry in the dictionary and the symbol twice; once in the dictionary and once for the actual block).

To decide whether it is useful to include an entry in the dictionary or not we also compare the entropy of the given block (in other words, the approximated number of bits the symbol for that block would have) to the addition of all the jpeg run length symbols required for the block. If the combination of jpeg symbols requires fewer bits than our symbol, putting it in the dictionary would be wasteful. This also has the advantage that by reducing the number of entries in the dictionary we will need fewer bits to encode the remaining symbols. Mathematically we only considered symbols that fulfill the following equation:

$$N \cdot H + J + H > H \cdot J$$

where:

N: Number of times block appears

H: Entropy of the symbol (rounded up to the closest larger integer)

J: Number of bits required for the jpeg encoding of the block

Once this has been done, we create a Huffman dictionary with the probability of appearance of each run/size block in the image as explained in 4.3.

We should mention that this expression does not guarantee gain for all the blocks that are entered in the dictionary. The reason for this is that if a coefficient is above threshold the sending of the difference is with our current scheme worse than jpeg's implementation, so even if we fulfill the above expression there might be particular situations where jpeg would have worked better than our encoding.

In some specific cases, with this expression we could also be removing symbols that could have remained in the dictionary. For example, if we remove all entries except for one, thus having a symbol for that entry and a default symbol, the entropy of the symbol might be high, but as there are only 2 entries in the dictionary we would only need 1 bit to send the encoded symbol, so we would most likely beat jpeg with our encoding.

6.4.- Optimization of jpeg encoding of high frequency coefficients (those not considered in the dictionary)

Taking into account that for AC coefficients different positions in the 8x8 DCT blocks have the same distribution but different variance [3], we considered that if we are to encode only some high-frequency coefficients using jpeg, the Huffman tables the scheme provides to encode the run/length of its AC coefficients might not be ideal. In order to ensure that we use the correct Huffman tables we created a python code that:

1) Obtains all the run/length combinations that appear in a testing set of 100 images given a specific threshold mask (the program will consider positions starting right after the last element in the threshold mask)

2) Organises all the run/length combinations by probability

3) Creates a Huffman dictionary from the probability of appearance of each run/size combination.

Results show the distribution starts to change if we consider a large number of coefficients in the dictionary (if we consider 8-10 coefficients the distribution is similar to considering the whole block).

6.5.- Optimised Huffman table for low frequency AC DCT coefficient sizes

For coefficients above threshold we have to send the size of difference from the coefficient to the threshold with a variable length symbol. Initially we were sending that symbol with jpeg's DC Huffman tables, but as the distribution of DC coefficients is different to that of AC coefficients it is clear that using those tables is not ideal. However, we still believe that creating a Huffman dictionary for the purpose is a good solution so, with that in mind we decided to optimize the dictionary by analysing the probability of appearance of each value above threshold for a given threshold and creating a Huffman table accordingly. Conceptually, the distribution of the difference is as follows:

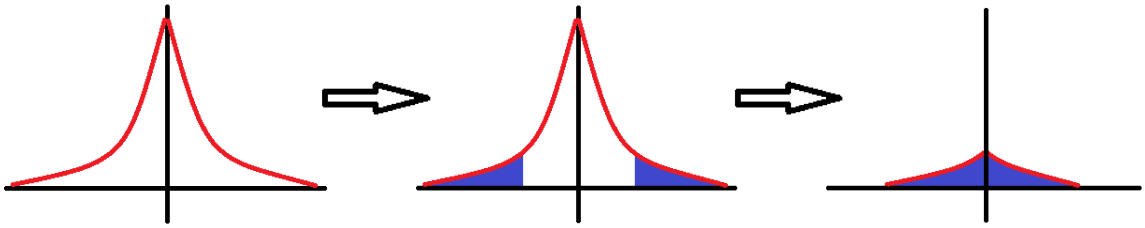

Fig. 23 – General distribution of AC coefficients (left). Distribution of difference after thresholding (right).

As we can see, the distribution of the difference between coefficients above threshold and threshold will still be Laplacian [3] but will have a much lower slope than the original distribution of AC coefficients.

In order to obtain our Hufman tables we developed a python program that does the following:

1) Given a specific threshold and threshold mask read all the coefficients above threshold (or below –threshold) from a test set of 100 images.

2) Find the difference between the coefficient and threshold (or –threshold)

3) Find the size in bits of the difference

4) Organise the sizes by probability of appearance

5) Create a Huffman dictionary from the probability of each size.

6.6.- Improved results

After adapting the improvements discussed in the previous sections we tested the modified implementation for image 1_baseline.jpg (Q=75). The x axis shows the number of times a block has to appear in the image to be considered for the dictionary and the y axis shows the size of the resulting files in Bytes (left) and the gain of the new scheme (public + private + dictionary) against jpeg (original).

With a Threshold mask of 10 coefficients with thresholds 16-8-4-2-2-1-1-1-1-1 and encoding the remaining 53 AC coefficients with jpeg we obtained the following:

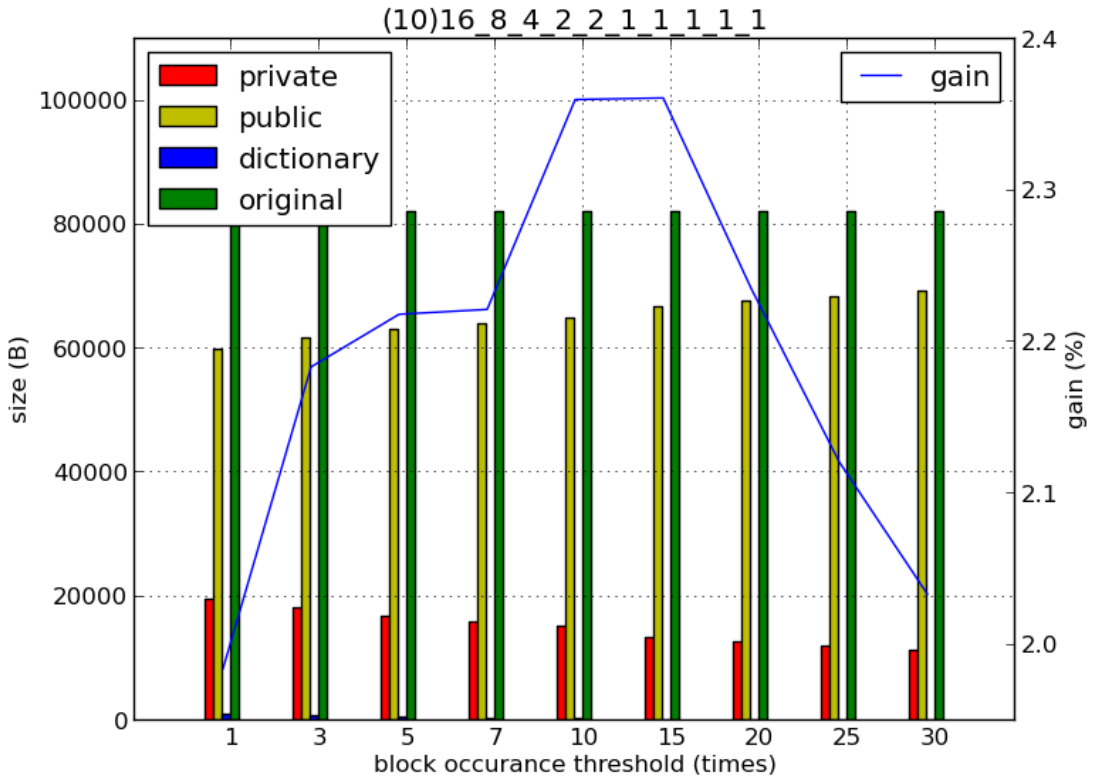

Fig. 24 – Gain new scheme – Thld mask: 10 coef. – img: 1_baseline.jpg.

As we can see, in this case we obtain some gain compared to the original jpeg file. However, the public file is still very close to the jpeg original file in size, which is an indicator that a lot of the blocks were not found in the dictionary (they were encoded with jpeg). As we explained, this could be caused by blocks not occurring enough times in the image or by the entropy of those blocks being higher than the cost of jpeg encoding the block.

We tried lowering the threshold values even more for the same threshold mask (2-2-1-1-1-1-1-1-1-1) and obtained the following:

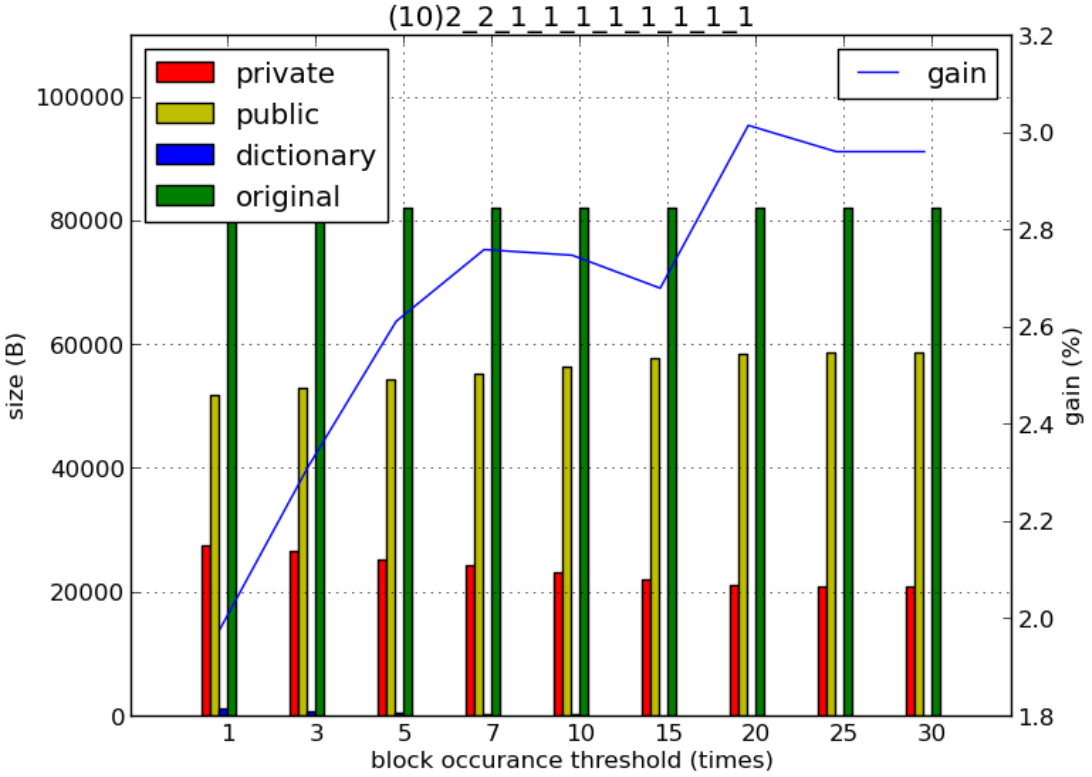

Fig. 25 – Gain new scheme – Thld mask: 10 coef. – img: 1_baseline.jpg.

For this case the public file is somewhat reduced, but as we threshold at lower values the private file also sees a considerable increase in size. We believe that our scheme works best with threshold values above 1 (with threshold one it is essentially as if we were only storing the location of nonzero coefficients). We will leave the analysis of different techniques for sending the private part for future work.

As for the fact that the public part is still similar in size to the original image, if we pay close attention to the dictionaries created for this case we will find the following:

If we analyse the symbol table for Y the component (with threshold 2-2-1-1-1-1-1-1-1-1):

```
16 0
3520 -1 -2
1401 0 2 0 2 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 -2
178 0 2 0 2 0 1 0 1 0 1 0 1 0 1 0 1 0 1 -2
119 0 2 0 2 0 1 0 1 0 1 1 1 0 1 0 1 0 1 -2
76 0 1 0 2 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 -2
63 0 2 0 2 0 1 0 1 0 1 0 1 1 1 0 1 0 1 -2
59 0 2 0 2 0 1 0 1 0 1 0 1 0 1 0 1 1 1 -2
59 0 2 0 2 0 1 0 1 0 1 0 1 0 1 1 1 0 1 -2
47 0 2 0 2 0 1 0 1 1 1 0 1 0 1 0 1 0 1 -2
43 0 2 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 -2
```

```
39 0 2 0 2 0 1 1 1 0 1 0 1 0 1 0 1 0 1 -2
39 0 2 0 2 0 1 0 1 0 1 1 1 0 1 0 1 -2
34 1 2 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 -2
34 0 2 0 2 0 1 0 1 0 1 0 1 0 1 0 1 -2
33 0 2 0 2 0 1 0 1 0 1 0 1 0 1 1 1 -2
32 0 2 0 2 1 1 0 1 0 1 0 1 0 1 0 1 0 1 -2
```

We can see how the only blocks that are introduced in the dictionary (those for which we get gain as defined in 6.3 are those with several non-zero coefficients that occur many times. The reason why we are not getting any gain for blocks that have few coefficients is that jpeg already performs very well for those cases and, as we are now considering only some of the coefficients to create the dictionary we have to send an EOB with each block, which might make the difference between gain and no gain for blocks with very few coefficients.

With this scheme the symbol table for the Cb and Cr components is empty, which means that when creating the dictionary the condition imposed to include a block in the dictionary is never met. Going back to what we discussed in 6.1, we believe we could get a better solution for Cb and Cr considering all the coefficients to create the dictionary. This means that we should use different thresholds and threshold masks for the different components (we will leave this to future work).

7.- Future Work

7.1.- Effects of varying the Scale Factor

As we mentioned previously, the quantization step in jpeg images has a user defined quality factor and, for smaller quality factors the 8x8 DCT coefficient blocks are divided by a larger number, thus being more likely to become 0. With that in mind we decided to analyse what happens for different threshold masks when changing the Quality factor from 75 to 50 (The same jpeg image 1_baseline.jpg had a size of 82Kb with Q=75 and 29.6Kb with Q=50) (Y axis: Number of blocks that appear “x” times, X axis: Number of appearances of block in image):

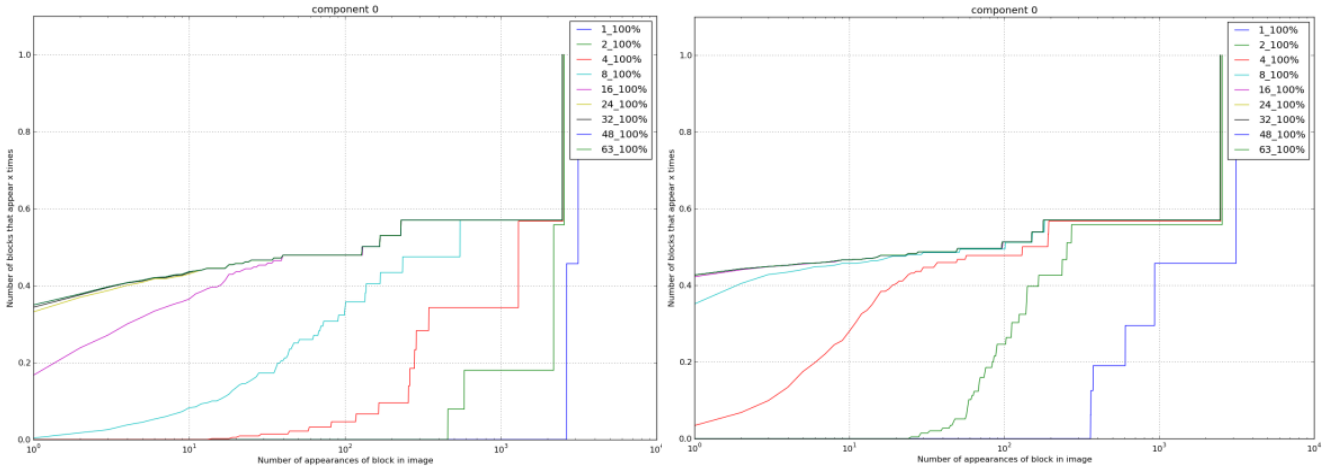

Fig. 26 – Block repetitions in Image 1_baseline.jpg – Thld = 1 (left) & 16 (right) - Q=50 – Component: Luminance (Y).

If we compare these results to what we found in 6.1 we will easily see a big difference in the block repetitions for big threshold masks. With a Quality factor of 50, results seem to indicate that our initial approach (where we considered all coefficients for the creation of the dictionary) might be better than jpeg. We should take into account that images with a lower quality factor are prone to have fewer and lower coefficients more concentrated around the low frequency locations, which means that it will be harder to beat jpeg. However, if we consider all the block when creating the dictionary our scheme also becomes more efficient (we don’t have to send EOB symbols or jpeg encoded information for coefficients after a specific location in the block) so we believe that with the reduced number of block combinations in the image (with threshold 16 and a threshold mask of 63 coefficients ~50% of the blocks occur more than 100 times) it is possible to perform better than jpeg.

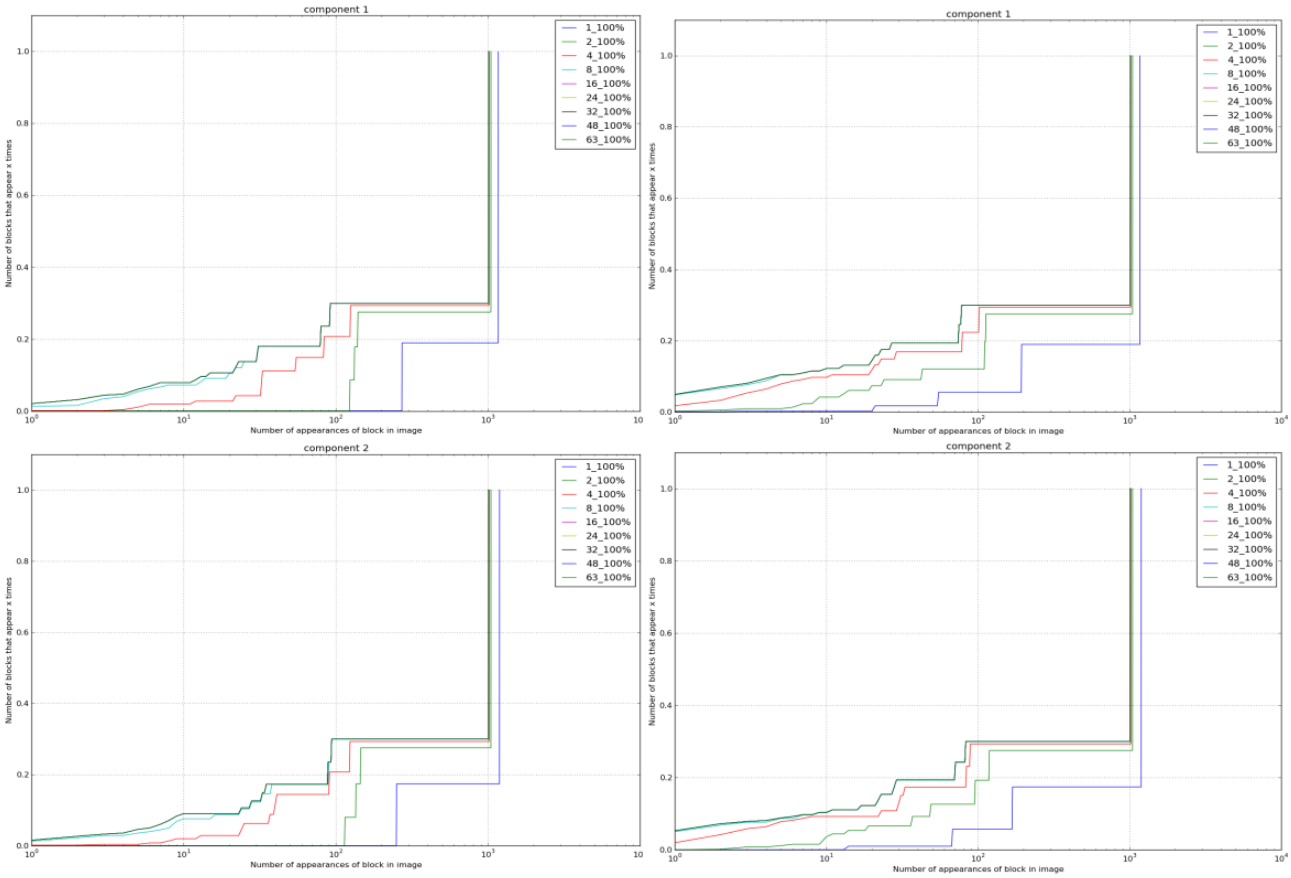

Fig. 27 – Block repetitions in Image 1_baseline.jpg – Thld = 1 (left) & 16 (right) - Q=50 – Component: Cb (top) & Cr (bottom).

For the Cb and Cr components we get very interesting results. As we discussed previously, these components are going to have even fewer and lower coefficient values than Luminance; however, in this case we believe that the number of combinations is so low and with so many occurrences of the same blocks (results seem to indicate that the "all zeros" block occurs more than 70% of the time even when considering the whole block) that it will still be possible to beat jpeg with our initial consideration. If we analyse the all zeros block for example, jpeg requires in this case only 2 bits to send the AC coefficients (EOB), but as it occurs so often our implementation would only need 1 bit for the same amount of information.

7.2.- Defining ideal threshold within "length" level - Pros and cons of different alternatives

As we explained in the implementation of the Encoder (5.1), we initially chose to create a system that takes advantage of the fact that if we choose a threshold value that is the lowest of the values in a specific length (fig. 10) we can avoid having to send extra information to tell the decoder whether a coefficient that has the same length as the threshold is below or equal/above threshold.

However, as we found out that our blocks were not common enough for the creation of the dictionary we started considering that perhaps this was not the wisest choice. The reason

being that, if instead of the lowest value in a specific size we choose the previous threshold value (the highest value in the previous size) we get two advantages:

1) The dictionary has a much lower number of possible combinations.

For example, if instead of using threshold 16 we use threshold 15, possible size combinations in the block go from 1,2,3,4,5 to 1,2,3,4, which means that our dictionary will become more common:

$$Combinations = (num\ sizes)^{64} \rightarrow 4^{64} < 5^{64}$$

It is important to note that with the distribution of AC coefficients we will get a much lower number of combinations for both cases (this is an upper bound).

2) For the same size choosing the highest value means that all the values before it can be sent without incurring in extra bits (remember that when we go above threshold we have to send extra information to know the size of the difference)

The disadvantage of this approach is, as we mentioned, that for coefficient values equal or above threshold as well as sending the size of the difference and the difference we would have to tell the encoder that the value associated to the size found in the block is equal or above threshold (we would need size(threshold) bits to do so). A small advantage being that we can send the sign of coefficients above threshold with those size(threshold) bits, which means that we could actually use fewer bits to send the difference. Proposed scheme:

-threshold < coef < threshold: send coefficient value (same as before)

coef == threshold or **coef == -threshold:** send coefficient value + symbol for size 0

coef > threshold or coef < -threshold: send threshold value (it tells us the sign) + symbol for size of abs(difference) + abs(difference)

We believe that this proposed scheme would perform better than the previous one for high threshold values and worse for very low threshold values (1 and 2). We leave a deeper analysis for future work.

7.3.- Analysis of the private part

As we previously mention, it is important to understand what is really happening with the coefficients that are above threshold for different cases, as we might find that the ideal case for dictionary creation is the worse in terms of performance of the private part with the current implementation, resulting in us losing possible gains.

In terms of private part, we believe that for very low thresholds our implementation performs poorly in comparison to jpeg (as we saw in 5.1 we have to send extra information for values above threshold); if the threshold is for example 1, we will have to send extra information for every non-zero coefficient. As we are currently finding that our best gains occur with low thresholds we feel it would be good to have a deeper understanding of the effects of different thresholds on the private part and different techniques to send the extra information.

7.4.- Delta encoding – Finding the closest block in the dictionary

In order to have less symbols with higher probability of appearance it would be interesting to select a set of blocks within an image that are very similar to other blocks within the image to, in case we don't find a specific block in the dictionary find the closest one and perform an encoding of the difference. With this method we could have fewer dictionary entries, resulting in shorter symbols, and could encode blocks that would otherwise occur very few times within the image.

As we are encoding the location and size of non-zero coefficients, differences across two blocks would be having different non-zero coefficient sizes for the same position. In other words, a block that had a 2 and a 3 in the same position would be considered the same (as 2 and 3 require the same number of bits to be sent), but a block with a 2 and a 6 in the same position would be considered to have 1 difference.

We could for example, cluster the blocks using Knearest neighbours being "nearest" the blocks that have the least number of different non-zero coefficient sizes. If we saw that blocks across images were similar enough we could find that this scheme provides good gains against jpeg, especially for blocks with many non-zero coefficients or rare combinations of values (each one might not be very likely but the addition of all of them accounts for a large part of the image data). We will leave this to future work.

7.5.- Common image dictionary

Using a common dictionary for several images instead of a per-image dictionary has its advantages and disadvantages. On one hand, a common dictionary for multiple images would mean that we don't have to send the overhead of a dictionary with every image. We can send one dictionary for a set of images or, ideally keep a dictionary common to all images in the encoder/decoder.

If we only considered this fact we would clearly use this approach; however, having a common dictionary for multiple images would mean that the symbols we use for each block are not optimized for each image, so we would be less efficient in our encoding. In addition, if different images don't share the same blocks or a similar distribution in the probability of appearance of those blocks, having a common dictionary would be more harmful than beneficial. If two images share 0 blocks and we created a common dictionary for both of them we would have a dictionary with double the size than if we created a dictionary for each one of them and as the dictionary would be twice as long, each symbol would require more bits to be sent. All in all, it would be a bad solution.

To summarize, this approach would work if two conditions are met:

1) Images share a high number of blocks

2) Those blocks have a similar probability of appearance across images.

To understand what would happen with our implementation we decided to study the evolution of the dictionary as we add more images. As we already saw for a single image that using all the coefficients to create the dictionary does not give a common enough set of blocks we decided to study the evolution for different threshold masks (Y axis: Number of different blocks in dictionary, X axis: Number of images in dictionary):

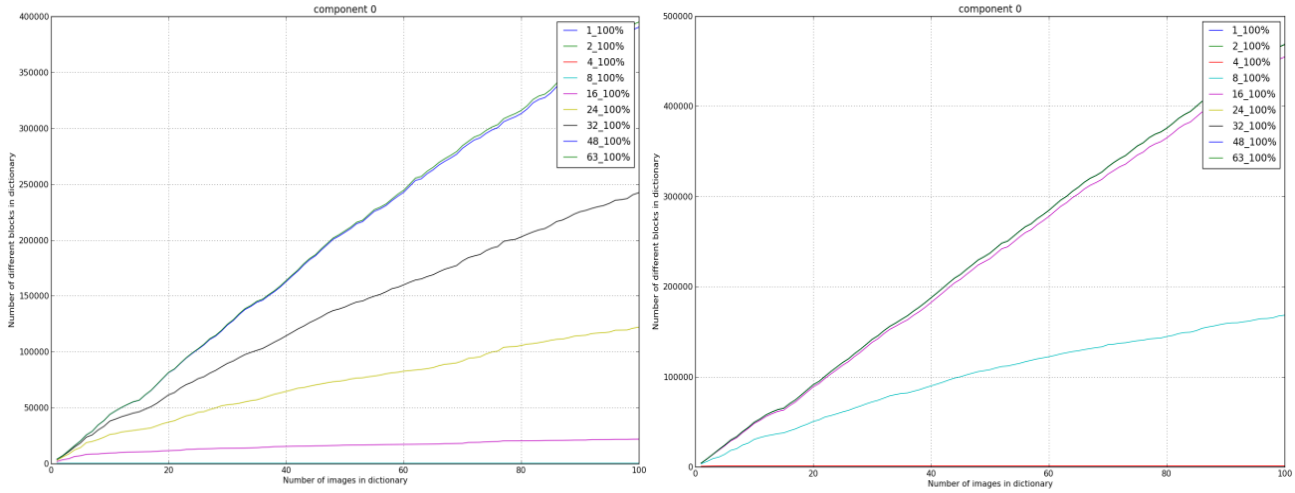

Fig. 28 – Dictionary growth – 100 img – Thld = 1 (left) & 16 (right) – Q=75.

For the luminance component we can see that even with a threshold of 1 (left), if we create a dictionary considering all the coefficients and combining several images, the dictionary grows

almost linearly, which is an indicator that the different images have few blocks in common. If we look at the number of appearances of specific blocks across the 100 images (11.2, fig. 32) we will find that if we consider all the coefficients, more than 60% of the blocks will occur only once in the 100 images, so it is not surprising to find this disparity in the dictionary growth. A threshold mask of around 16 coefficients has a much more reasonable growth for this component. This is, however, for threshold 1; if we look at the case for threshold 16 the growth of the dictionary has a as we expected a steeper slope. In this case, with a threshold mask of 8 coefficients we find that around 80% of the blocks occur more than once and about 50% more than 10 times.

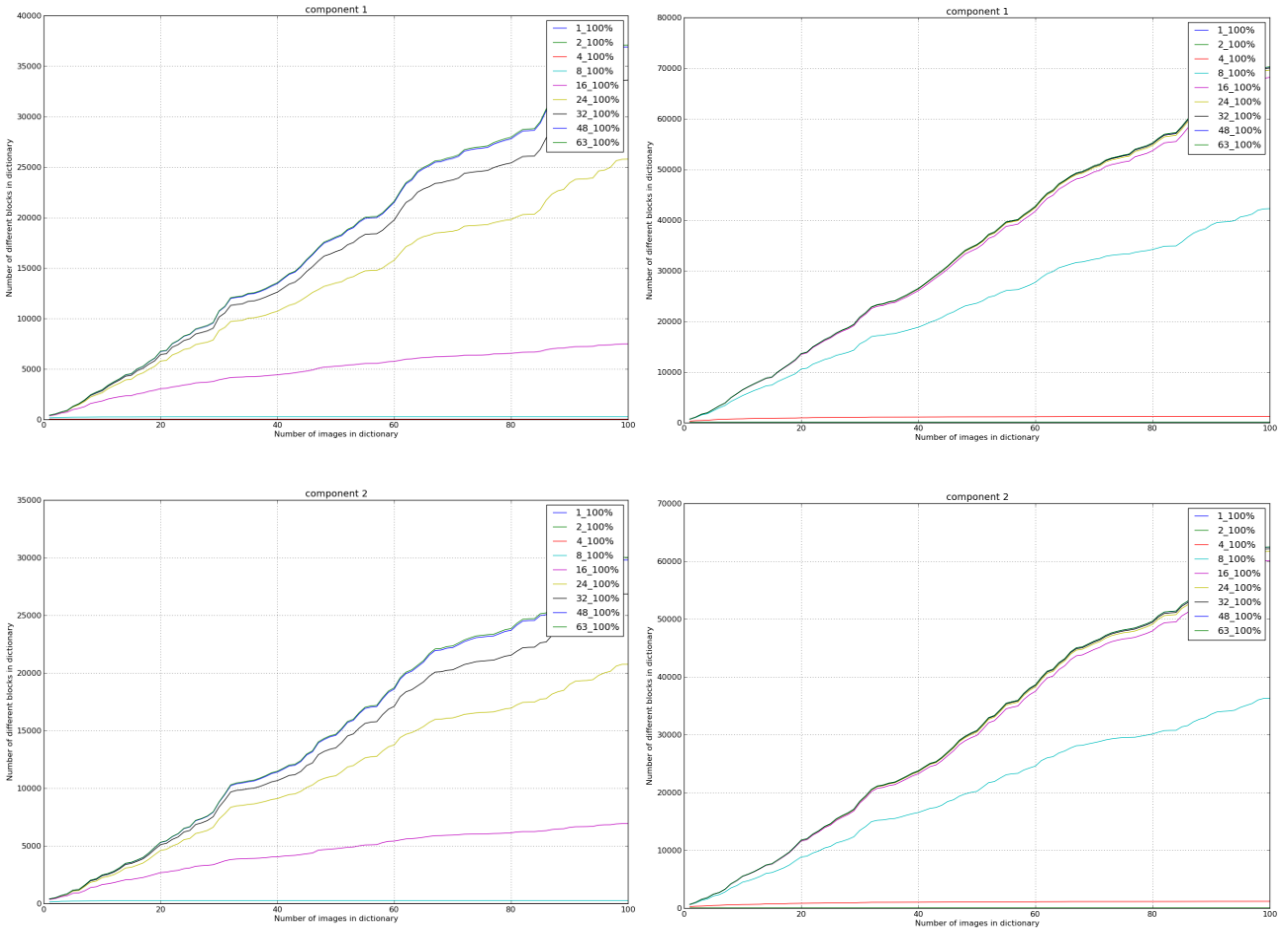

Fig. 29 – Dictionary growth 100 img for Cb (up) & Cr (bottom) – Thlds: 1 (left) & 16 (right) – Q=75.

For Cb and Cr there is a much lower number of combinations than for Y, but the growth of the dictionary for several images still has a steep slope. These results seem to indicate that creating a common dictionary for several images with this scheme is not going to result in large gains when compared to jpeg. However, if we manage to make the blocks across images more common (with delta encoding, for example) it could become very promising.

8.- Conclusions

After all the analysis and tests we carried out, we can say that the techniques used in the Privacy Preserving Photo Algorithm developed at the University of Southern California sets grounds to create a new encoding scheme for images that results in better compression gains than jpeg.

At this point, our implementation obtains compressed images that are approximately 3% smaller than jpeg's when creating a per-image dictionary. However, we believe that by finding the optimal combination of variables in our scheme (for example choosing the ideal: thresholds for each coefficient, threshold mask, condition to introduce elements in the dictionary) and/or finding a way to make blocks within and across images more common (with delta encoding, for example), we can obtain substantially larger compression gains than jpeg.

9.- Acknowledgements

The completion of this project would not have been possible without the never ending guidance and support provided by Professor Antonio Ortega, who gave me multiple ideas and advice and has been involved in the project from beginning to end. I would also like to thank Professor Ramesh Govindan and PhD student Zahaib Akhtar for their support and involvement in the project and for all the time spent analysing ideas and results both in our weekly meetings and at other times. Finally, I would like to thank PhD student Xing Xu for the countless hours spent helping with the development of the implementation and working on different ideas for the overall development of the project. Thank you all for your help!

10.- Bibliography

[1] Moo-Ryong Ra, Ramesh Govindan, Antonio Ortega, *P3: Toward Privacy-Preserving Photo Sharing*, University of Southern California

[2] Gregory K. Wallace, *The JPEG Still picture compression Standard*, December 1991

[3] Edmund Y. Lam, Joseph W. Goodman, *A Mathematical Analysis of the DCT Coefficient Distributions for Images*, IEEE Transactions on Image Processing, Vol. 9, Nº 10, October 2000

[4] Bernd Girod, *EE398 JPEG standard no. 1 - Image and Video Compression*, Stanford

[5] Mr.S. V. Viraktamath, Dr. Girish V. Attimarad, *Impact of Quantizatino Matrix on the Performance of JPEG,* International Journal of Future Generation Communication and Networking Vol. 4, N. 3, September, 2011

[6] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, Peter Vajgel, *Finding a needle in Haystack: Facebook's photo storage*, Facebook Inc.

[7] International Telecommunication Union, *Information Technology – Digital Compression and coding of continuous-tone still images – Requirements and guidelines*, September 1992

[8] Ying-Wun Huang, Yi-Fan Chang, *An implementation for JPEG decoder,* Graduate Institute of Communication Engineering College of Electrical Engineering and Computer Science National Taiwan University

11.- Annex

11.1.- Images

Fig. 30 – Image: Coffee cup.jpg

Fig. 31 – Image 1_baseline.jpg

11.2.- Additional figures

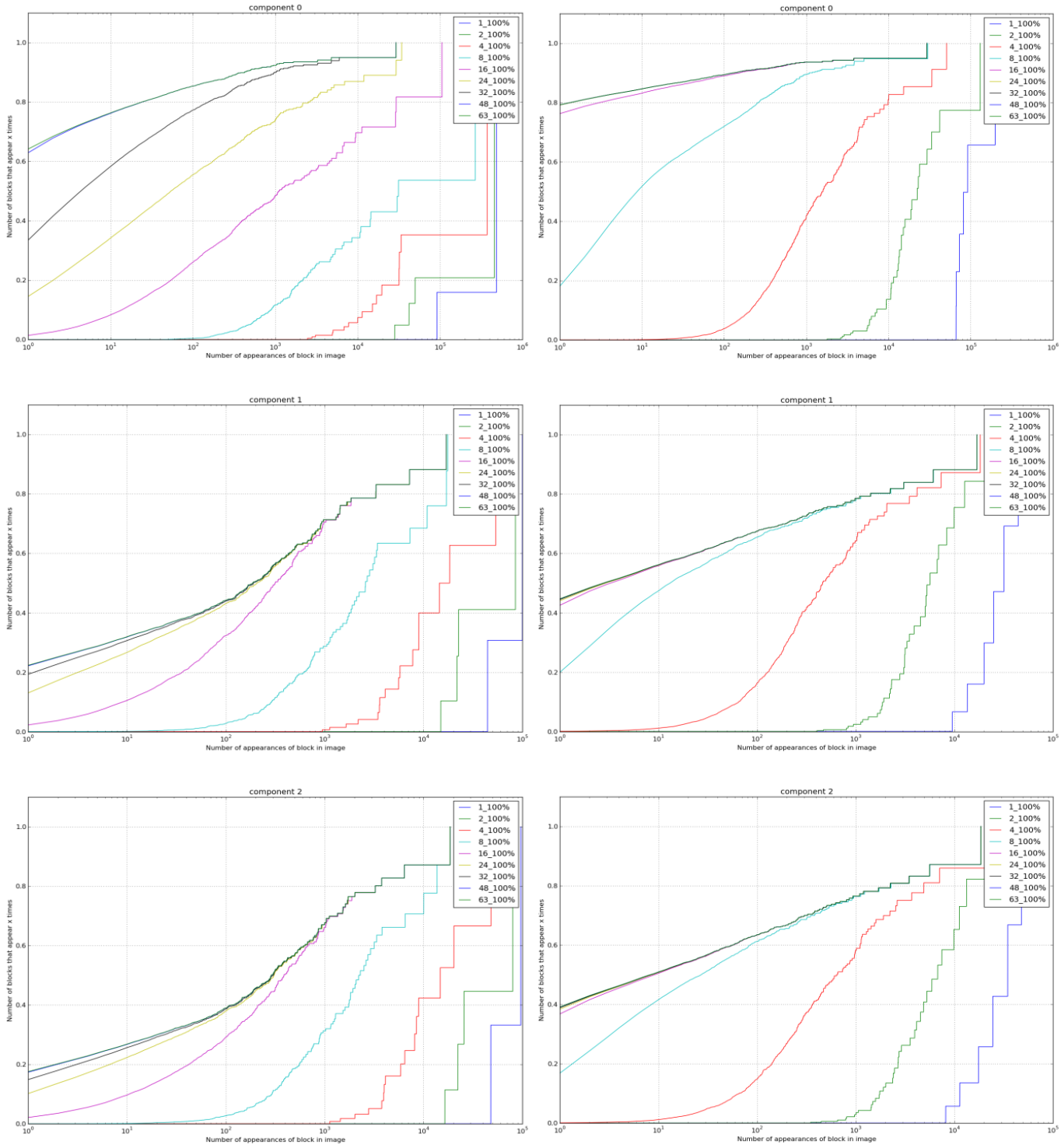

Fig. 32 –Block repetitions for different thld masks – 100 images – Thld = 1 (left) & 16 (right) - Q=75 – Components: Y (top), Cb (middle), Cr (bottom) (Y axis: Number of blocks that appear "x" times, X axis: Number of appearances of block in image).

PFC

Index:

1.- Introduction

1.1.- Roadmap

2.- Background

2.1.- JPEG

2.2.- P3 Algorithm

2.3.- Libjpeg

3.- Motivation

3.1.- Visual and size effects of different thresholds on public image

3.2.- Effects of maintaining the sign of coefficients above threshold in the public part

3.5.- Finding the ideal threshold in terms of image quality and image size:

3.6.- Practical implementation of the P3- Algorithm:

3.7.- Different approach ideas:

3.7.1.- All blocks encoding

3.7.2.- Nonzero encoding

3.7.4.- Hybrid encoding

4.- Implementation

4.1. Encoder

4.1.2.- Detailed encoding of the private part:

4.2.- Decoder

4.3.- The Dictionary

4.4.- Results initial approach:

5.- Improvements initial implementation

5.1.- Creating a dictionary for low frequency coefficients and encoding the rest with JPEG:

5.2.- Using different thresholds for different coefficients in the image:

5.3.- Decision of what blocks to introduce in the dictionary

5.3.- Optimization of “jpeg-like” encoding of bits not considered in the dictionary:

5.4.- Optimised Huffman table for low frequency AC DCT coefficient sizes:

5.5.- Results improved implementation

6.- Future work

6.1.- Lowering the Scale Factor:

6.2.- Different thresholds and threshold masks for Y-Cb-Cr

6.3.- Defining ideal threshold within size range - Pros and cons of different alternatives.

6.4.- Analysis of private part -> what is happening under the hood?

6.5.- Delta encoding – Finding the closest block in the dictionary:

6.6.- Common image dictionary:

7.-Special thanks

8.- Bibliography

Initial P3 Obtained images study:

In order to study modifications and different situations for the P3 algorithm we implemented several functions in Matlab:

Functions:

```
function Dq = jpeg_compression(im_ycbcr)
%JPEG_COMPRESSION This function obtains the 8x8 DCT coefficients of an
% image from the Y-Cb-Cr representation
% INPUT: Y-Cb-Cr decomposition of image
% OUTPUT: 8x8 DCT coefficients of image (for the 3 components)

function ycbcr_out = jpeg_decompression(Dq, N_fil, N_col)
% JPEG_DECOMPRESSION This function recovers the Y-Cb-Cr representation
of the image
% INPUT:
%       - Dq: 8x8 DCT coefficients
%       - N_fil: height of the image in pixels
%       - N_col: width of the image in pixels
% OUPUT:
%       - ycbcr_out: Y-Cb-Cr representation of the image


function  [Dq_public,  Dq_private]  =  p3_algorithm(Dq_public,  N_fil,
N_col, threshold)
%P3_ALGORITHM This function separates the original image into public
%and secret images (always in 8x8 DCT coefficients)
% INPUT:
%           - Dq_public: 8x8 DCT coefficients for Y, Cb & Cr
%           - N_fil: height of the image in pixels
%           - N_col: width of the image in pixels
%           - threshold: Threshold value
% OUTPUT:
%           - Dq_public: 8x8 DCT coefficients for the public image
%           - Dq_private: 8x8 DCT coefficients for the secret image


function  Dq_recomposed  =  p3_recomposition(Dq_public,  Dq_private,
N_fil, N_col, threshold)
%P3_RECOMPOSITION This function recovers the 8x8 DCT coefficients of
%the original image
% INPUT:
%           - Dq_public: 8x8 DCT coefficients for the public image
%           - Dq_private: 8x8 DCT coefficients for the secret image
%           - N_fil: Height of the image in pixels
%           - N_col: Width of the image in pixels
%           - threshold: Threshold value
% OUTPUT:
%           - Dq_recompoosed: 8x8 DCT coefficients of original image
```

[x] Annex 1: Matlab Code

Annex 1: Matlab Code:

jpeg_compression:

```
function Dq = jpeg_compression(im_ycbcr)
%JPEG_COMPRESSION This function obtains the 8x8 DCT coefficients of an
% image from the Y-Cb-Cr representation
% INPUT: Y-Cb-Cr decomposition of image
% OUTPUT: 8x8 DCT coefficients of image (for the 3 components)

% Determine whether to quantize: (Yes = 1, No = 0) (With No we still
round up Dq)
quantize = 1;
scale_factor = 1;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%

%Zig-Zag order:
zig_zag_order = [1 9 2 3 10 17 25 18 11 4 5 12 19 26 33 41 34 27 20 13
6 7 14 21 28 35 42 49 57 50 43 36 29 22 15 8 16 23 30 37 44 51 58 59
52 45 38 31 24 32 39 46 53 60 61 54 47 40 48 55 62 63 56 64];


% Quantization matrix.
Q_y = [16,11,10,16,24,40,51,61; 12,12,14,19,26,58,60,55;
    14,13,16,24,40,57,69,56; 14,17,22,29,51,87,80,62;
    18,22,37,56,68,109,103,77; 24,35,55,64,81,104,113,92;
    49,64,78,87,103,121,120,101; 72,92,95,98,112,100,103,99];


Q_cb_cr = [17,  18,  24,  47,  99,  99,  99,  99,
    18,  21,  26,  66,  99,  99,  99,  99,
    24,  26,  56,  99,  99,  99,  99,  99,
    47,  66,  99,  99,  99,  99,  99,  99,
    99,  99,  99,  99,  99,  99,  99,  99,
    99,  99,  99,  99,  99,  99,  99,  99,
    99,  99,  99,  99,  99,  99,  99,  99,
    99,  99,  99,  99,  99,  99,  99,  99];

% % Separate YCbCr images.
% y = im_ycbcr(:,:,1);
% cb = im_ycbcr(:,:,2);
% cr = im_ycbcr(:,:,3);

% Determine image matrix size.
N_fil = size(im_ycbcr(:,:,1),1);
N_col = size(im_ycbcr(:,:,1),2);
size_A = (N_fil/8)*(N_col/8); %Si no es entera la division redondea,
trunca o que hace¿?
Dq = zeros(8,8,size_A,3);
zig_zag = zeros(64, size_A, 3);

for p=1:3
```

```
    A = ones(8,8,size_A);
    % Generate 8x8 matrixes
    for i=1:N_fil/8
        for j=1:N_col/8
            for k=1:8
                for l =1:8
                    A(k,l,(N_col/8)*(i-1)+j)    =    im_ycbcr(8*(i-1)+k,
8*(j-1)+l, p);
                end
            end
        end
    end


    % Extract DC coefficients from A.
    Shift = ones(8,8,size_A)*128;
    A = A - Shift;


    % Get DCT Matrix and Block by block DCT
    D_mtx = dctmtx(8);
    D = zeros(8,8,size_A);
    %     M = zeros(size_A);
    for m=1:size_A
        D(:,:,m) = D_mtx*A(:,:,m)*D_mtx';
    end

    % Quantization of DCT.
    if quantize == 1
        if p == 1
            Q = Q_y*scale_factor;
        else
            Q = Q_cb_cr*scale_factor;
        end
        % Quantization + rounding of D (DCT).

        for i=1:size_A
            Dq(:,:,i,p) = D(:,:,i)./Q;
        end
        Dq = round(Dq);
    else
        Dq(:,:,:,p) = round(D);
    end

end
end
```

p3_algorithm:

```
function [Dq_public, Dq_private] = p3_algorithm(Dq_public, N_fil,
N_col, threshold)
%P3_ALGORITHM This function separates the original image into public
%and secret images (always in 8x8 DCT coefficients)
% INPUT:
%           - Dq_public: 8x8 DCT coefficients for Y, Cb & Cr
%           - N_fil: height of the image in pixels
%           - N_col: width of the image in pixels
%           - threshold: Threshold value
% OUTPUT:
%           - Dq_public: 8x8 DCT coefficients for the public image
%           - Dq_private: 8x8 DCT coefficients for the secret image

size_A = N_fil/8 * N_col/8;
Dq_private = zeros(8,8,size_A,3);

for p=1:3
    % We put the DC components in the private image.
    for i=1:size_A
        Dq_private(1,1,i,p) = Dq_public(1,1,i,p);
        Dq_public(1,1,i,p) = 0;
        % Values above threshold are sent to private image.
        for k=1:8
            for l=1:8
                if Dq_public(k,l,i,p) > threshold
                    Dq_private(k,l,i,p)    =    Dq_public(k,l,i,p)    -
threshold;
                    Dq_public(k,l,i,p) = threshold;
                elseif Dq_public(k,l,i,p) < (-threshold)
                    Dq_private(k,l,i,p)    =    Dq_public(k,l,i,p)    +
threshold;
                    Dq_public(k,l,i,p) = threshold;
                end
            end
        end
    end
end
end
```

p3_recomposition:

```
function Dq_recomposed = p3_recomposition(Dq_public, Dq_private,
N_fil, N_col, threshold)
%P3_RECOMPOSITION This function recovers the 8x8 DCT coefficients of
%the original image
% INPUT:
%           - Dq_public: 8x8 DCT coefficients for the public image
%           - Dq_private: 8x8 DCT coefficients for the secret image
%           - N_fil: Height of the image in pixels
%           - N_col: Width of the image in pixels
%           - threshold: Threshold value
% OUTPUT:
%           - Dq_recompoosed: 8x8 DCT coefficients of original image

size_A = N_fil/8 * N_col/8;
Dq_recomposed = zeros(8,8,size_A, 3);
for p=1:3
    for i=1:size_A
        for j=1:8
            for k=1:8
                if Dq_public(j,k,i,p) == threshold
                    if Dq_private(j,k,i,p)>=0
                        Dq_recomposed(j,k,i,p) = Dq_private(j,k,i,p) +
Dq_public(j,k,i,p);
                    else
                        Dq_recomposed(j,k,i,p) = Dq_private(j,k,i,p) -
Dq_public(j,k,i,p);
                    end
                else
                    Dq_recomposed(j,k,i,p) = Dq_public(j,k,i,p);
                end
            end
        end
        Dq_recomposed(1,1,i,p) = Dq_private(1,1,i,p);
    end
end
end
```

jpeg_decompression:

```
function ycbcr_out = jpeg_decompression(Dq, N_fil, N_col)
% JPEG_DECOMPRESSION This function recovers the Y-Cb-Cr representation
of the image
% INPUT:
%       - Dq: 8x8 DCT coefficients
%       - N_fil: height of the image in pixels
%       - N_col: width of the image in pixels
% OUPUT:
%       - ycbcr_out: Y-Cb-Cr representation of the image

Scale_factor = 1;

size_A = N_fil/8*N_col/8

% Quantization matrix.
Q_y = [16,11,10,16,24,40,51,61; 12,12,14,19,26,58,60,55;
    14,13,16,24,40,57,69,56; 14,17,22,29,51,87,80,62;
    18,22,37,56,68,109,103,77; 24,35,55,64,81,104,113,92;
    49,64,78,87,103,121,120,101; 72,92,95,98,112,100,103,99];

Q_cb_cr = [17,  18,  24,  47,  99,  99,  99,  99,
    18,  21,  26,  66,  99,  99,  99,  99,
    24,  26,  56,  99,  99,  99,  99,  99,
    47,  66,  99,  99,  99,  99,  99,  99,
    99,  99,  99,  99,  99,  99,  99,  99,
    99,  99,  99,  99,  99,  99,  99,  99,
    99,  99,  99,  99,  99,  99,  99,  99,
    99,  99,  99,  99,  99,  99,  99,  99];

% Used at the end of the function.
ycbcr_out_bis = zeros(N_fil, N_col, 3);

for p=1:3
    if p == 1
        Q = Q_y*scale_factor;
    else
        Q = Q_cb_cr*scale_factor;
    end
    %Decompression
    D1 = zeros(8,8,size_A);
    for i=1:size_A
        D1(:,:,i) = Dq(:,:,i,p).*Q;
    end

    % IDCT + add DCT coefficients.
    A1 = zeros(8,8,size_A);
    for i=1:size_A
        A1(:,:,i) = idct2(D1(:,:,i));
    end
    Shift = ones(8,8,size_A)*128;
    A1 = A1 + Shift;

    % Recover y matrix.
    % Al principio del codigo esta la inicializacion de ycbcr_out_bis.
    for i=1:N_fil/8
        for j=1:N_col/8
            for k=1:8
                for l =1:8
```

```
                    ycbcr_out_bis(8*(i-1)+k,8*(j-1)+l,        p)        =
A1(k,l,(N_col/8)*(i-1)+j);
                end
            end
        end
    end
end
% Convert ycbc_out values to uint8.
ycbcr_out = uint8(ycbcr_out_bis);
end
```

Annex xx: analysis_blocks_single_image.py

```
# step1, we get blocks in run_length type, with specified
threshold
# step2, use the block file in step 1, to generate the symbol
table
# step3, encode using that symbol table
# step4, decode and check if it's correct

import os, sys, operator, math
from pylab import *
import numpy


folder = "../../images/webp_600x600_img/"
files  =  ['1_baseline.jpg','2_baseline.jpg',  '3_baseline.jpg',
'4_baseline.jpg',       '5_baseline.jpg',       '6_baseline.jpg',
'7_baseline.jpg',       '8_baseline.jpg',       '9_baseline.jpg',
'10_baseline.jpg']
#thresholds = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20]
#threshold_bits = [1, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4,
5, 5, 5, 5, 5]
thresholds = [1, 2, 4, 8, 16]
threshold_bits = [1, 2, 3, 4, 5]
entries = ['100%']
thld_masks = [1,2,4,8,16,24,32,48,63]
plot_folder                                                    =
"../../results/plot_results/cdf_analysis_webp_img_01/"

table_size = 0

def calc(filename, dic, tot, times):
      lines = open(filename).readlines()
      for i in range(len(lines)):
            s = lines[i]
            temp = s.split(": ")
            comp = int(temp[0])
            keys = temp[1].split("-2")
            key = keys[0] + "-2"
            if (key in dic[comp]):
                  dic[comp][key] += int(keys[1])
                  times[comp][key] += 1
            else:
                  dic[comp][key] = int(keys[1])
                  times[comp][key] = 1
            tot[comp] +=  1
      return dic, tot, times

def get_distribution(file_, entries, thld_mask):
      percentage = False
      if entries.find("%")>0:
            percentage = True
            entries = int(entries[:-1])
      else:
            percentage = False
```

```
            entries = int(entries)

      #threshold = int(sys.argv[3])
      #threshold_bits = int(sys.argv[4])

      os.system("mkdir " + str(plot_folder))

      files = file_.split(",")

      dic = []
      dic.append({})
      dic.append({})
      dic.append({})
      times = []
      times.append({})
      times.append({})
      times.append({})

      tot = []
      tot.append(0)
      tot.append(0)
      tot.append(0)

      f = []
      f.append(open("temp_0", 'w'))
      f.append(open("temp_1", 'w'))
      f.append(open("temp_2", 'w'))

      for i in range(len(files)):
            dic, tot, times = calc(files[i], dic, tot, times)

      #f_size = open("symbol_table_size.txt", "w")

      ret = []
      for i in range(3):
            filename = "cdf_image_comp_" + str(i) + "_thld_mask_"
+ str(thld_mask)
            print filename + ": "
            a             =             sorted(dic[i].iteritems(),
key=operator.itemgetter(1), reverse=True)
            max_reps          =          max(times[i].iteritems(),
key=operator.itemgetter(1))[1] + 1
            count = [0]*max_reps
            num_bars = times[i][a[0][0]]
            jj = 0
            f_list = []
            upper_bound = len(a)

            temp  = 0
            for j in range(upper_bound):
                  if times[i][a[j][0]] > max_reps:
                        print   "ERRORRRRRR   "   +   "pos   0:   "   +
str(times[i][a[0][0]])     +     "          |otherone:      "     +
str(times[i][a[j][0]])
            #     f_list.append((a[j][0], times[i][a[j][0]]))
            #     a[j][1]/times[i][a[j][0]]
```

```
                count[times[i][a[j][0]]] += times[i][a[j][0]]
                temp += times[i][a[j][0]]

          for j in range(len(count)):
                count[j]/=1.0*temp
                if j:
                      count[j]+=count[j-1]
          ret.append(count)
     #    a    =    sorted(f_list,    key=operator.itemgetter(1),
reverse=True)
     #    for j in range(upper_bound):
     #    ind = np.arange(len(num_bars))
     print ret
     return ret

#root_folder = sys.argv[1]
#os.system("mkdir " + root_folder)
for ff in files:
     f = folder + ff
     baseline_size = os.path.getsize(f)
     print f + ":"
     fig, axarr = plt.subplots(3, figsize=(16,36))
     for i in range(3):
          axarr[i].set_ylabel('Number of blocks that appear x
times')
          axarr[i].set_xlabel('Number of appearances of block
in image')
          axarr[i].set_title('component ' + str(i))
          axarr[i].set_ylim((0, 1.1))
          axarr[i].grid()
          axarr[i].set_xscale('log')
     legend = []
     for thld_mask in thld_masks:
          file_thld = open("thresholds.txt","w")
          file_thld.write("0 ")
          for i in range(thld_mask):
                file_thld.write("1 ")
          for i in range((63-thld_mask)):
                file_thld.write("0 ")
          file_thld.close()

          step1 = "jpegtran -outputcoef " + f + "_" +
str(thld_mask) + " 1 " + f + " temp > temp2"
#               print step1 + "!!!"
          os.system(step1)
          for entry in entries:
                ret_vals = get_distribution(f + "_" +
str(thld_mask), str(entry), str(thld_mask))
                legend.append(str(thld_mask) + "_" + str(entry))
                for j in range(3):
                      count = ret_vals[j]
                      axarr[j].plot(range(len(count)), count)
     for i in range(3):
          axarr[i].legend(legend, 1)
     savefig(plot_folder + ff + ".png")
```

Annex xxxx: make_table.py

This python code makes a Huffman table with the run/size blocks of the input files (one file => per-image dictionary, multiple files => common dictionary)

```
# make symbol table
# python make_table.py file_1,file_2,file_3 -portion 100 // do
100% entries
# python make_table.py file_1,file_2,file_3 -entry 100 // do 100
entries
# python make_table.py file_1,file_2,file_3 -threshold 2 // do
entries that occurs at least 2 times

import os, sys, operator, math

percentage = False
threshold = -1
entries = 1e20

file_ = sys.argv[1]
option = sys.argv[2]
number = sys.argv[3]

if option.find("portion")>0:
      percentage = True
      entries = int(number)
if option.find("entry")>0:
      entries = int(number)
if option.find("threshold")>0:
      threshold = int(number)

files = file_.split(",")

dic = []
dic.append({})
dic.append({})
dic.append({})
times = []
times.append({})
times.append({})
times.append({})

tot = []
tot.append(0)
tot.append(0)
tot.append(0)

f = []
f.append(open("temp_0", 'w'))
f.append(open("temp_1", 'w'))
f.append(open("temp_2", 'w'))


def calc(filename):
      lines = open(filename).readlines()
```

```
      for i in range(len(lines)):
            s = lines[i]
            temp = s.split(": ")
            comp = int(temp[0])
            keys = temp[1].split("-2")
            key = keys[0] + "-2"
            if (key in dic[comp]):
                  dic[comp][key] += int(keys[1])
                  times[comp][key] += 1
                  dic[comp][key] -= 2
                  if comp:
                        dic[comp][key] -= 2
            else:
                  dic[comp][key] = int(keys[1])
                  times[comp][key] = 1
                  dic[comp][key] -= 2
                  if comp:
                        dic[comp][key] -= 2
            tot[comp] +=  1

for i in range(len(files)):
      calc(files[i])

f_size = open("symbol_table_size.txt", "w")
table_size = 0
for i in range(3):
      a = sorted(dic[i].iteritems(), key=operator.itemgetter(1),
reverse=True)
      temp_c = 0
      f_list = []
      gain = 0
      max_ = len(a)
      if percentage:
            upper_bound = min(max_, entries*max_/100)
      else:
            upper_bound = min(max_, entries)

      for j in range(upper_bound):
#           print "bits :", a[j][1]/times[i][a[j][0]], "gain: ",
a[j][1], "times: ", times[i][a[j][0]], "run block: ", a[j][0]
            if times[i][a[j][0]] > threshold:     # only consider
the block occurs more than threshold times
                  temp_c += times[i][a[j][0]]
                  prob = times[i][a[j][0]]*1.0/tot[i]
                  entropy = math.ceil(-math.log(prob, 2))
                  gain += a[j][1] - entropy*times[i][a[j][0]]
                  f_list.append((a[j][0], times[i][a[j][0]]))
                  a[j][1]/times[i][a[j][0]]
      default_times = max(1, tot[i] - temp_c)
#     print "gain=", gain, "loss=", default_times, "diff (B)=",
(gain-default_times)/8.0
#     print f_list
      f_list.append(('-1 -2', default_times))
      a     =     sorted(f_list,     key=operator.itemgetter(1),
reverse=True)
```

```
      f[i].write(str(len(f_list)) + " " + str(a.index(('-1 -2',
default_times))) + "\n")
      for j in range(len(a)):
#           print a[j][1], a[j][0]
            temp = a[j][0].split("-2")
            f[i].write(str(a[j][1]) + " " + temp[0] + "-2 \n")
      f[i].close()
#     print  "python  ../huff_coding/prob_to_table.py  temp_"  +
str(i) + " symbol_table" + str(i) + ".txt"
      os.system("python ../huff_coding/prob_to_table.py temp_" +
str(i) + " symbol_table" + str(i) + ".txt")
      f_dict = open("symbol_table" + str(i) + ".txt").readlines()
      for j in range(1, len(f_dict)):
            symbol_bits = int(f_dict[j].split(" ")[0])
            run_length_str = f_dict[j][f_dict[j].find(" ") + 1:-
2]
            if run_length_str in dic[i]:
                  run_length_bits                                 =
dic[i][run_length_str]/times[i][run_length_str]
            else:
#                 print "not found" + run_length_str
                  run_length_bits = 0
#           print    run_length_str,    ":",    run_length_bits,
symbol_bits
            table_size += symbol_bits + run_length_bits
f_size.write(str(table_size/8.0))
```

Annex xxxx:

Python code called by make_table.py and make_high_freq_table.py to create a Huffman dictionary from the probabilities of appearance:

```
import sys, huff

input_file = sys.argv[1]
output_file = sys.argv[2]

input_lines = open(input_file).readlines()
out_file = open(output_file, "w")

num = int(((input_lines[0]).split(" "))[0])
out_file.write(input_lines[0])

prob = []
for i in range(1, num + 1):
      data = input_lines[i].split(" ")
      prob.append(float(data[0]))

codes = huff.huffman(prob)

bits = {}
for i in range(len(prob)):
      bits[i] = len(codes[i])

for i in range(1, num + 1):
      out_file.write(str(bits[i - 1]))
      out_file.write(input_lines[i][input_lines[i].find(" "):])
out_file.close()
```